NI MATRIXx™

Xmath™ Control Design Module

April 2007
370753C-01

Worldwide Technical Support and Product Information

`ni.com`

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599,
Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00,
Finland 385 (0) 9 725 72511, France 33 (0) 1 48 14 24 24, Germany 49 89 7413130, India 91 80 41190000,
Israel 972 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400,
Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466,
New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210,
Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197,
Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222,
Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at `ni.com/info` and enter the info code `feedback`.

© 2007 National Instruments Corporation. All rights reserved.

Important Information

Warranty

TThe media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

Trademarks

MATRIXx™, National Instruments™, NI™, ni.com™, and Xmath™ are trademarks of National Instruments Corporation.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

<> — Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, DIO<3..0>.

[ ] — Square brackets enclose optional items—for example, [`response`].

» — The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.

This icon denotes a note, which alerts you to important information.

bold — Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic — Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

`monospace italic` — Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Contents

Chapter 1 Introduction

Using This Manual..1-1
Document Organization..1-1
Bibliographic References..1-2
Commonly Used Nomenclature..1-2
Related Publications..1-3
MATRIXx Help..1-3
Control Design Tutorial..1-4
Helicopter Hover Problem: An Ad Hoc Approach..1-4
Helicopter Hover Problem: State Feedback and Observer Design..1-13
Helicopter Hover Problem: Discrete Formulation..1-18
Inverted Wedge-Balancing Problem: LQG Control..1-20

Chapter 2 Linear System Representation

Linear Systems Represented in Xmath..2-1
Transfer Function System Models..2-2
State-Space System Models..2-5
Basic System Building Functions..2-6
system( )..2-6
abcd( )..2-8
numden( )..2-10
period( )..2-10
names( )..2-11
Size and Indexing of Dynamic Systems..2-12
Using check( ) with System Objects..2-12
Discretizing a System..2-13
discretize( )..2-13
Numerical Integration Methods: forward, backward, tustins..2-14
Pole-Zero Matching: polezero..2-15
Z-Transform: ztransform..2-15
Hold Equivalence Methods: exponential and firstorder..2-15
makecontinuous( )..2-17

Chapter 3
Building System Connections

Linear System Interconnection Operators 3-1
Linear System Interconnection Functions 3-4
afeedback( ) 3-4
Algorithm 3-5
append( ) 3-6
connect( ) 3-8
Algorithm 3-10
feedback( ) 3-11
Algorithm 3-12

Chapter 4
System Analysis

Time-Domain Solution of System Equations 4-1
System Stability: Poles and Zeros 4-2
poles( ) 4-3
zeros( ) 4-3
Algorithm 4-5
Partial Fraction Expansion 4-5
residue( ) 4-8
combinepf( ) 4-9
General Time-Domain Simulation 4-10
Impulse Response of a System 4-13
impulse( ) 4-13
deftimerange( ) 4-15
System Response to Initial Conditions 4-16
initial( ) 4-16
Step Response 4-18
step( ) 4-18

Chapter 5
Classical Feedback Analysis

Feedback Control of a Plant Model 5-1
Root Locus 5-2
rlocus( ) 5-3
Frequency Response and Dynamic Response 5-5
freq( ) 5-5
Algorithm 5-6

Bode Frequency Analysis 5-7
bode( ) 5-10
margin( ) 5-12
nichols( ) 5-14
Nyquist Stability Analysis 5-15
nyquist( ) 5-16
Linear Systems and Power Spectral Density 5-20
psd( ) 5-20

Chapter 6
State-Space Design

Controllability 6-1
controllable( ) 6-3
Observability and Estimation 6-4
observable( ) 6-6
Minimal Realizations 6-7
minimal( ) 6-8
stair( ) 6-9
Duality and Pole Placement 6-10
poleplace( ) 6-10
Linear Quadratic Regulator 6-12
regulator( ) 6-14
Linear Quadratic Estimator 6-16
estimator( ) 6-20
Linear Quadratic Gaussian Compensation 6-21
lqgcomp( ) 6-23
Riccati Equation 6-25
riccati( ) 6-26
Steady-State System Response Using Lyapunov Equations 6-28
lyapunov( ) 6-30
Algorithm 6-30
rms( ) 6-32
Balancing a Linear System 6-33
balance( ) 6-35
Modal Form of a System 6-37
modal( ) 6-37
mreduce( ) 6-38

Appendix A
Technical References

Appendix B
Technical Support and Professional Services

Index

1 Introduction

The Control Design Module (CDM) is a complete library of classical and modern control design functions that provides a flexible, intuitive control design framework.

This chapter starts with an outline of the manual and some user notes. It also contains a tutorial that presents several problems and uses a variety of approaches to obtain solutions. The tutorial is designed to familiarize you with many of the functions in this module.

Using This Manual

This manual provides an overview of different aspects of linear systems analysis, describes the Xmath Control Design function library, and gives examples of how you can use Xmath to solve problems rapidly. It also explains how you can represent and analyze linear systems in Xmath and provides a brief syntax listing and supplementary algorithm information for each CDM function. Detailed descriptions of function inputs, outputs, and behavior are provided in the *Xmath Help*.

Document Organization

This manual includes the following chapters:

- Chapter 1, *Introduction*, starts with an outline of the manual and some user notes. It also contains a tutorial that presents several problems and uses a variety of approaches to obtain solutions. The tutorial is designed to familiarize you with many of the functions in this module.
- Chapter 2, *Linear System Representation*, describes the types of linear systems that can be represented within Xmath. In addition, it discusses the implementation of systems as objects–data structures encompassing different information fields. The Xmath functions for creating a system or extracting its components are part of the general Xmath package and not exclusive to the Control Design Module, but they are used so extensively that they warrant a detailed treatment here. This chapter also discusses the functions you can use to check for

particular system properties or to change the format of a system. These topics include continuous/discrete system conversion, as well as finding equivalent transfer function state-space representations.

- Chapter 3, *Building System Connections*, details Xmath functions that perform different types of linear system interconnections. It also discusses a number of simpler connections that have been implemented as overloaded operators on system objects.
- Chapter 4, *System Analysis*, describes the Xmath functions relating to system stability and time-domain analysis. These include poles, zeros, and residue. The chapter moves from the discussion of time-domain stability to time-domain system simulation. Xmath provides built-in functions for obtaining impulse and step responses, as well as examining system response to arbitrary initial conditions. In addition, the *General Time-Domain Simulation* section discusses a mathematically natural syntax for time-domain system simulation with any input.
- Chapter 5, *Classical Feedback Analysis*, discusses topics pertaining to classical feedback-based control design. These include root locus techniques and functions for frequency-domain analysis of closed-loop systems, given open-loop system descriptions.
- Chapter 6, *State-Space Design*, focuses on modern control. Beginning with the topics of system controllability and observability, it covers general pole placement, linear quadratic control, and system balancing.

Bibliographic References

Throughout this document, bibliographic references are cited with bracketed entries. For example, a reference to [DeS74] corresponds to a document published by Desoer and Schulman in 1974. For a table of bibliographic references, refer to Appendix A, *Technical References*.

Commonly Used Nomenclature

This manual uses the following general nomenclature:

- Matrix variables are generally denoted with capital letters; vectors are represented in lowercase.
- $G(s)$ is used to denote a transfer function of a system where s is the Laplace variable. $G(q)$ is used when both continuous and discrete systems are allowed.

- *H*(*s*) is used to denote the frequency response, over some range of frequencies of a system where *s* is the Laplace variable. *H*(*q*) is used to indicate that the system can be continuous or discrete.
- A single apostrophe following a matrix variable, for example, *x*', denotes the transpose of that variable. An asterisk following a matrix variable (for example, *A**) indicates the complex conjugate, or Hermitian, transpose of that variable.

Related Publications

For a complete list of MATRIXx publications, refer to Chapter 2, *MATRIXx Publications, Online Help, and Customer Support*, of the *MATRIXx Getting Started Guide*. The following documents are particularly useful for topics covered in this manual:

- *MATRIXx Getting Started Guide*
- *Xmath User Guide*
- *Control Design Module*
- *Interactive Control Desing Module*
- *Interactive System Identification Module, Part 1*
- *Interactive System Identification Module, Part 2*
- *Model Reduction Module*
- *Optimization Module*
- *Robust Control Module*
- *Xμ Module*

MATRIXx Help

Control Design Module function reference information is available in the *MATRIXx Help*. The *MATRIXx Help* includes all Control Design functions. Each topic explains a function's inputs, outputs, and keywords in detail. Refer to Chapter 2, *MATRIXx Publications, Online Help, and Customer Support*, of the *MATRIXx Getting Started Guide* for complete instructions on using the Help feature.

Control Design Tutorial

This tutorial illustrates the use of functions and commands provided in Xmath and the Xmath Control Design Module to solve control problems. The emphasis of the tutorial is on using a number of different approaches, not on any one “correct” way to solve a problem. It demonstrates the flexibility of Xmath’s tools and scripting language to customize your analysis in a way that is as straightforward and mathematically intuitive as possible.

The models in this tutorial are adapted from the studies in [ShH92], of the equations presented in [FPE87], for the longitudinal motion of a helicopter near hover, and in [HW91], for the inverted-wedge-balancing problem.

Helicopter Hover Problem: An Ad Hoc Approach

[FPE87] gives this state-space model for the longitudinal motion of the helicopter:

$$\begin{bmatrix} \dot{q} \\ \dot{\theta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} -0.4 & 0 & -0.01 \\ 1 & 0 & 0 \\ -1.4 & 9.8 & -0.02 \end{bmatrix} \begin{bmatrix} q \\ \theta \\ v \end{bmatrix} + \begin{bmatrix} 6.3 \\ 0 \\ 9.8 \end{bmatrix} \delta$$

$$y = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q \\ \theta \\ v \end{bmatrix}$$

letting the state variables q, θ, and v represent the helicopter’s pitch rate, pitch angle, and horizontal velocity, respectively. The input control to the system is the rotor tilt angle, δ.

You can store the information that this model provides in an Xmath state-space system object:

```
A = [-0.4,0,-0.01;1,0,0;-1.4,9.8,-0.02];
B = [6.3;0;9.8];
C = [0,0,1];
D = 0;
```

```
ssys = system(A,B,C,D,
    {inputNames ="Rotor Angle",
    outputNames="Horizontal v",
    stateNames =["Pitch Rate", "Pitch Angle",
    "Horizontal v"]})
```

```
ssys (a state space system) =

A
-0.4   0   -0.01
 1     0    0
-1.4   9.8 -0.02

B
6.3
0
9.8

C
0    0    1

D
0

X0
0
0
0

State Names
-----------
Pitch Rate     Pitch Angle     Horizontal v

Input Names
-----------
Rotor Angle

Output Names
------------
Horizontal v
System is continuous
```

Use `check( )` to convert the model to transfer-function form:

```
[,Gs] = check(ssys,{tf,convert})

Gs (a transfer function) =

       2
9.8(s - 0.5s + 6.3)
-----------------------------------------
              2
(s + 0.656513)(s - 0.236513s + 0.149274)

initial integrator outputs
0
0
0
```

```
Input Names
-----------
Rotor Angle

Output Names
------------
Horizontal v

System is continuous
```

The system has poles and zeros in the right half of the complex plane and therefore is open-loop unstable. Checking the pole and zero locations confirms this:

```
ol_poles=poles(ssys)

ol_poles (a column vector) =
   0.118256 - 0.367817 j
   0.118256 + 0.367817 j
   -0.656513

ol_zeros=zeros(ssys)

ol_zeros (a column vector) =
   0.25 + 2.4975 j
   0.25 - 2.4975 j
```

Try to stabilize the system using feedback compensation. You have two major performance goals to achieve through your controller design: first, the system must be closed-loop stable, and second, you want the system output to track a unit step input. To begin, put two compensators in the feedforward path of the closed-loop system. Figure 1-1 is a closed-loop block diagram of helicopter system $H(s)$ with compensators $K_1(s)$ and $K_2(s)$ in the feedforward path.

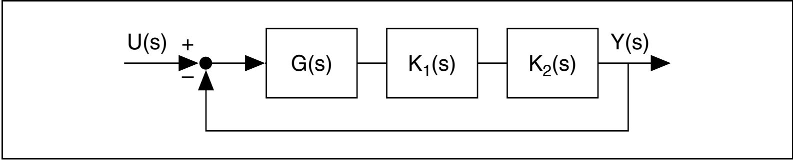


Figure 1-1. Block Diagram of Helicopter System $H(s)$ with Compensators $K_1(s)$ and $K_2(s)$ in the Feedforward Path

One approach to stabilizing this system is to try to cancel the pole at –0.656513 by adding a compensator, $K_1(s)$, with a zero at –0.656513.

Note It is important to understand that this is primarily an academic exercise. Accurate pole-zero cancellations are impracticable in the real world, and the mode corresponding to that pole still exists internally.

This compensator must have a pole for realizability, so you add one at –10, which is far enough away that its effect on dynamic response will be small compared to that of the system's other modes. In addition, you need to add a zero to the left of the positive (and unstable) poles to pull the closed-loop system roots into the left half plane. Choose $s = 0$ for the zero location and, again, select a corresponding pole at –10. Call this second compensator $K_2(s)$. To create these two compensators:

```
K1s=polynomial(ol_poles(3),"s")/...
    polynomial(-10,"s");
K2s=polynomial(0,"s")/polynomial(-10,"s");
```

You then can cascade them in series with the original system `Gs` (or `ssys`) and examine the locus of closed-loop roots for varying total compensator gain K_c. The poles out at –10 have a smaller effect on the system dynamics than do the poles closer to the origin, so you can use the optional `rlocus( )` keywords to zoom in on the part of the locus nearer the origin.

```
rlocus(K2s*K1s*Gs, {xmin = -2, xmax = 2})
```

The single input syntax activates interactive mode. A user interface lets you change the gain through the **Feedback Loop Gain** slider and button. The graphics window shows the closed-loop locus as a solid line, with the open-loop poles shown as large `x`s and the open-loop zeros shown as `O`s. Increase the gain by moving the slider; notice the asterisks (`*`) denoting closed-loop pole location moving along the locus. The system is maximally stable with total compensator gain $K_c = 2$ as shown in Figure 1-2. In this figure, small `x`s denote the pole location for $K_c = 2$, and the root locus gain window shows settings.

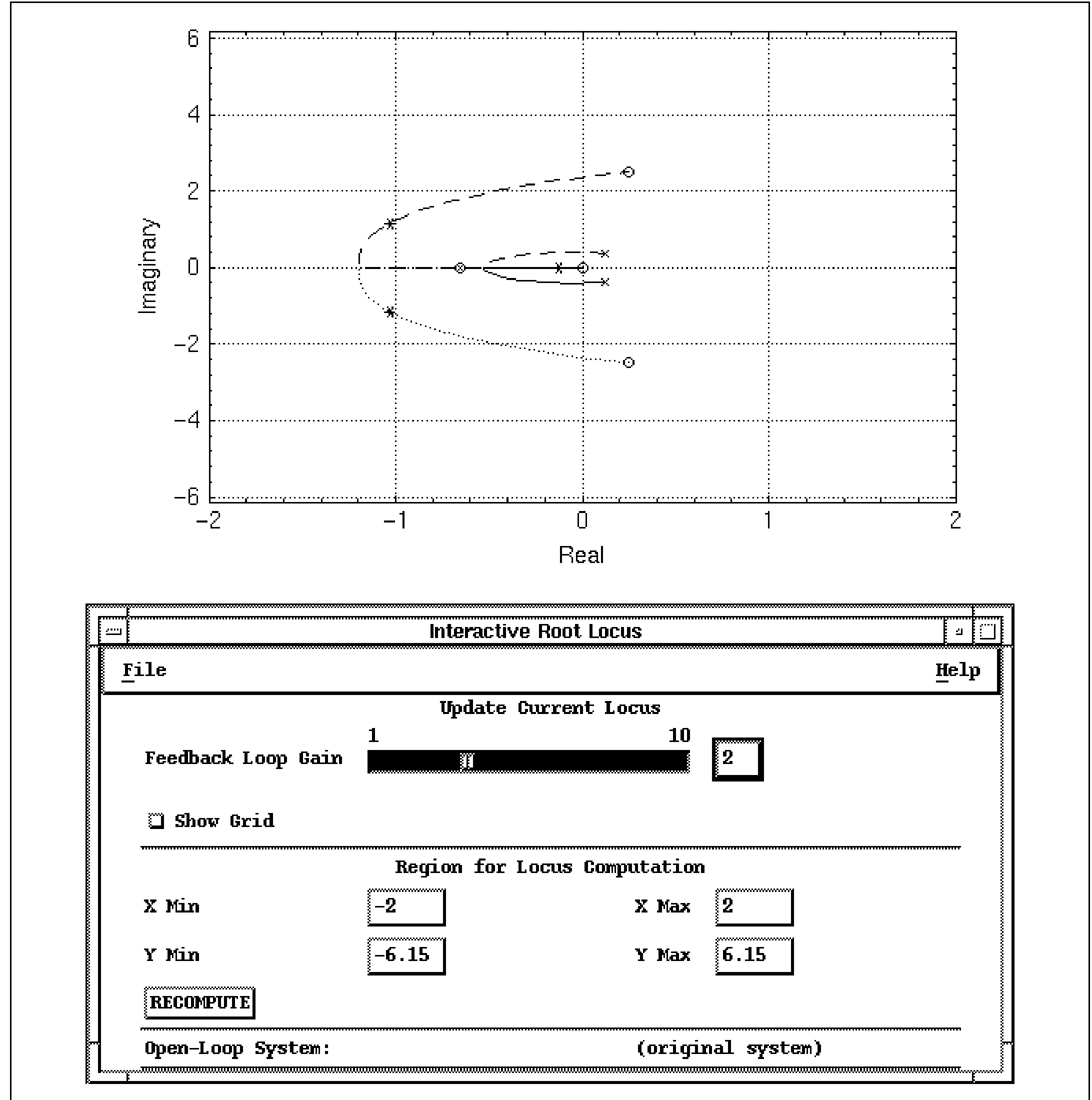


Figure 1-2. Locus of all Open-Loop and Closed-Loop Roots of Gs

If you cannot move the slider so that the gain is exactly 2, click the box to the right of the slider and enter `2`. To close the interactive root locus dialog box, select **File»Exit**.

Close the loop using the single-input syntax of `feedback( )`, which implements direct unity-gain negative feedback, and obtain the system's step response using `step( )`:

```
Kc = 2; cl_syscomp1 = feedback(Kc*K1s*K2s*Gs);
v = step(cl_syscomp1, 0:.2:25);
plot(v,{xlab="Time",       ylab="Horizontal Velocity"})
```

The resulting plot is shown in Figure 1-3. This result is not desirable. You want the output (the helicopter velocity) to track the step input provided as the rotor tilt angle, not zero out its effects over time (which would be an appropriate response if the input corresponded to a disturbance). This results from the compensator zero at $s = 0$ in the forward path of the feedback loop.

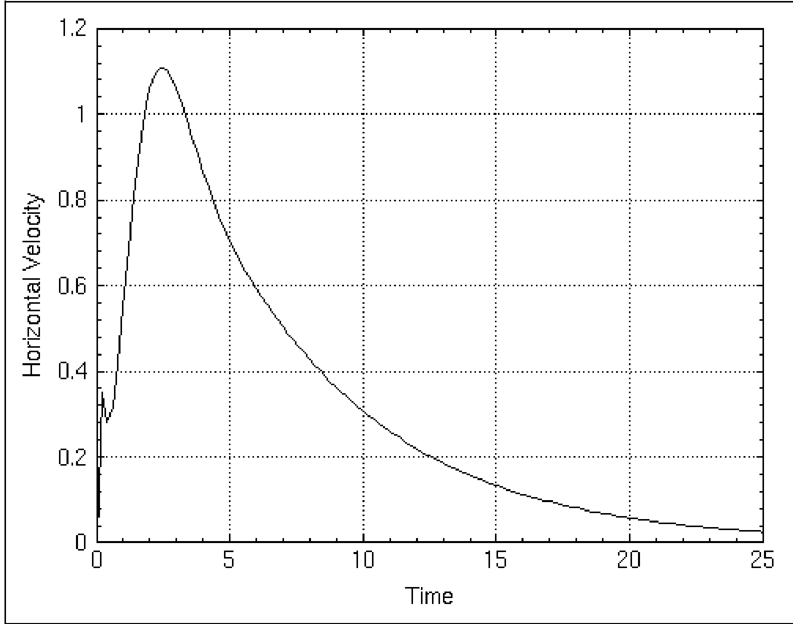


Figure 1-3. Helicopter Velocity Response to a Step Input at the Rotor

Instead, you now place $K_1(s)$ in the forward path and $K_2(s)$ in the feedback path, so that the closed-loop system now has the configuration shown in Figure 1-4.

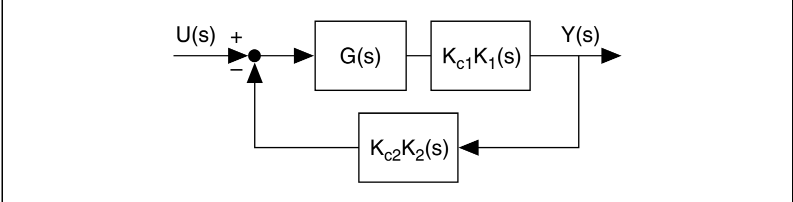


Figure 1-4. Block Diagram of the Closed-Loop Controller

This is a block diagram of the closed-loop controller with compensator $K_{c1}K_1(s)$ in the feedforward path and $K_{c2}K_2(s)$ in the feedback path. This time, instead of having all your gain K_c in the forward path of the closed-loop system, the system gain is split between the two compensators. The gains K_{c1} and K_{c2} are defined such that $K_c = 2 = K_{c1}K_{c2}$ and the closed-loop transfer function $T_{c1}(s)$ is unity at $s = 0(DC)$.

The closed-loop transfer function is represented by:

$$T_{cl}(s) = \frac{K_{c1}K_1(s)G(s)}{1 + K_{c1}K_{c2}K_1(s)K_2(s)G(s)}$$

You can find the values of the individual transfer functions at $s = 0$ using `freq( )`, and then substitute to solve the previous equation:

```
a = makematrix(freq(K1s*Gs,0));
b = makematrix(freq(K1s*K2s*Gs,0));
```

Solving:

```
Kc1 = (1+2*b)/a

Kc1 (a scalar) = 0.0241778

Kc2 = 2/Kc1

Kc2 (a scalar) = 82.7206
```

You now call `feedback( )` again, this time using its second input argument to indicate that the outputs of the first input system (forward path) are fed back as the inputs to the second system (feedback path) in a negative-feedback loop.

```
cl_syscomp2 = feedback(Kc1*K1s*Gs, Kc2*K2s);
```

Because `cl_syscomp2` contains an internal pole-zero cancellation, you can rewrite it in minimal form and then check the closed-loop pole and zero locations:

```
cl_syscomp2m = minimal(cl_syscomp2);
```

```
The system has 1 uncontrollable state
```

```
cl_poles = poles(cl_syscomp2m)
```

```
cl_poles (a column vector) =

   -0.166518
   -1.0325 + 1.16109 j
   -1.0325 - 1.16109 j
   -37.132
```

```
cl_zeros = zeros(cl_syscomp2m)
```

```
cl_zeros (a column vector) =
   0.25 - 2.4975 j
   0.25 + 2.4975 j
   -10
```

Now, examine the step response as shown in Figure 1-5.

```
vcomp = step(cl_syscomp2m, 0:.1:25);
plot (vcomp, {xlab = "Time",
   ylab = "Horizontal Velocity"})
```

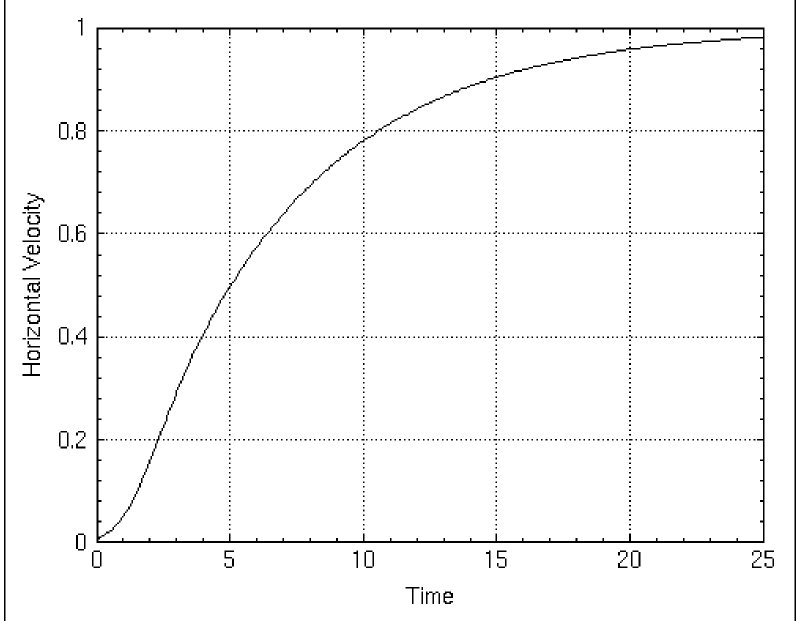


Figure 1-5. Helicopter Velocity Tracking Step Input at the Rotor

You also can look at the gain and phase margins of the system.

```
H = bode(cl_syscomp2m, {npts = 200, !wrap});
[gm,pm] = margin(H)

There are no 0 dB gain crossings.
gm (a pdm) =

domain |
---------+----------
0.250101 | 26.1694
---------+----------

pm is null
```

The bode plot of the closed-loop system is shown in Figure 1-6.

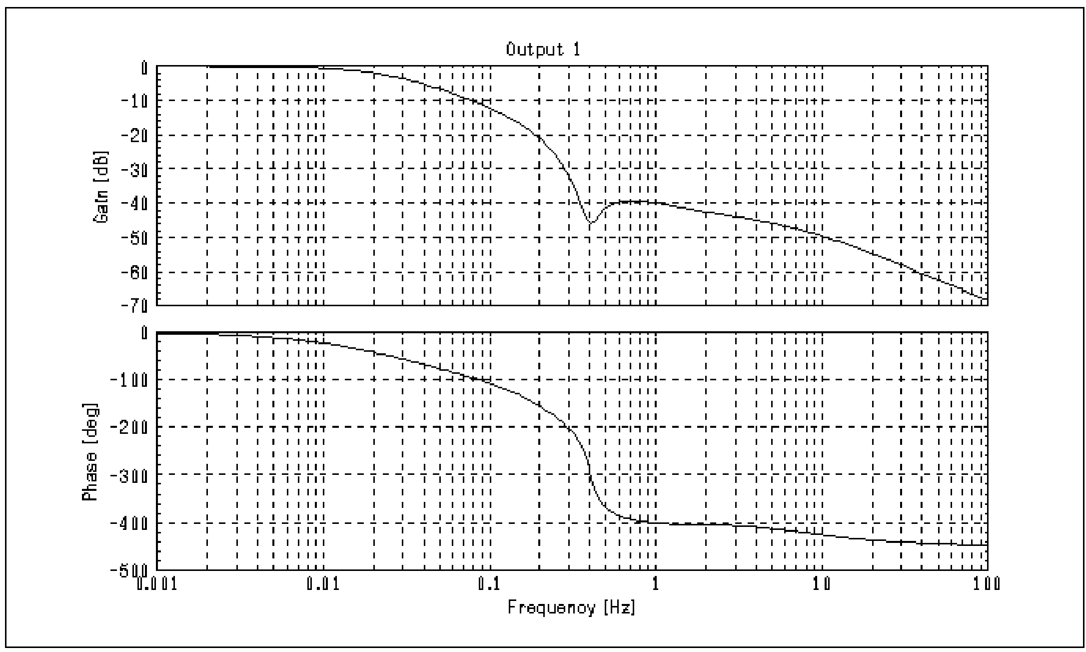


Figure 1-6. Closed-Loop System Bode Plot

The domain of the gain and phase margin PDMs indicates the frequency (in hertz) at which the margin occurs. So the gain can be increased by about 26.1 dB before the system becomes unstable.

Helicopter Hover Problem: State Feedback and Observer Design

The approach taken in the *Helicopter Hover Problem: An Ad Hoc Approach* section, although producing a desirable response, often cannot be used in practice because uncertainty in modeling generally precludes exact knowledge of the location of the pole one plans to cancel.

Another approach is to feed the information obtained from the states back to the inputs through gains calculated to relocate the closed-loop poles. Refer to the *Controllability* section of Chapter 6, *State-Space Design*, for more information. For this approach, you first need to verify that your system is controllable and observable. When you have confirmed that it is—that there are no hidden modes—you can design a full-state feedback control law that will place the system eigenvalues at values that will yield a stable system. Because the system is observable, you then can design an estimator to yield estimates for the missing states. Again, you will require that your system track a step input.

You can verify that your system is controllable, then define the closed-loop poles you want and use `poleplace( )` to find the feedback gains required given the system A and B matrices.

```
[,,nuc] = controllable(Gs)
```

```
nuc (a scalar) = 0
```

Because the number of uncontrollable states is zero, `Gs` is controllable. This means that you can use feedback through appropriately-sized gains to position the system's closed-loop poles anywhere you want. If you choose the three poles to be moved to –1 ±*j* and –2, you get the following set of gains:

```
clp = [-1+jay, -2];
Kfsb = poleplace(A,B,clp)
```

```
Kfsb (a row vector) = 0.470648 1.00004 0.062747
```

Note `poleplace( )` does not require you to list both poles in a conjugate pair.

If you assume that the outputs of the system are just the values of all the states, you can draw the open-loop system block diagram as shown in Figure 1-7. In this figure, the feedback path is shown in dotted lines and the open-loop system in solid lines.

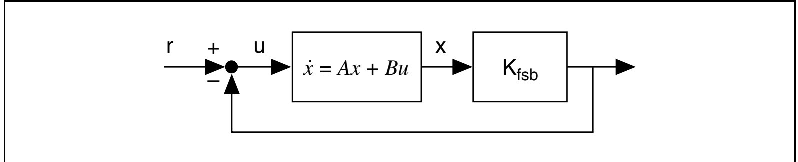


Figure 1-7. Full-State Feedback Regulator

Because you do not have access to all three states—only one, the horizontal velocity, is returned as an output—you need to estimate the other states, thus implementing an observer-based controller. The block diagram for the observer and controller together is shown in Figure 1-8.

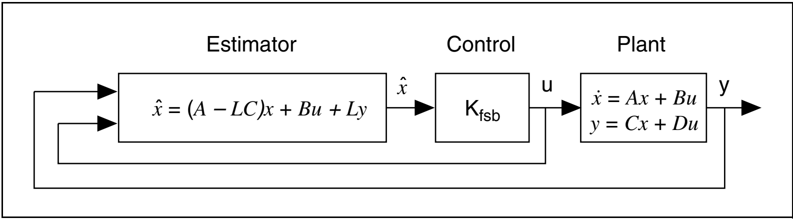


Figure 1-8. Complete Controller and Estimator

Specify the observer poles at [–3 ± 3j, –4] and call `poleplace( )` again:

```
op = [-3+3*jay, -4];
```

```
L = poleplace(A',C',op)
```

```
L (a row vector) = 5.46645   4.67623   9.58
```

You connect the controller to the observer using `lqgcomp( )`. `L` needs to be a column vector, so you transpose it.

```
sys_obc = lqgcomp(ssys, Kfsb, L');
```

You can use `names( )` to modify the names of the state estimates to be more descriptive. To distinguish the estimated states from the “true” states, you can use the `+` operator to append the string `est` to the estimated state names, as shown in this example.

```
[,,osNames] = names(sys_obc);
estNames = osNames + [" (est)"," (est)"," (est)"];
estNames'?
```

```
ans (a column vector of strings) =
Pitch Rate (est)
Pitch Angle (est)
Horizontal v (est)
```

You can append these modified names to `sys_obc`:

```
sys_obc = system(sys_obc,{stateNames=estNames});
```

Then you close the loop and verify that the closed-loop poles are all in the left-half plane.

```
sys_cl = feedback(ssys, sys_obc);
```

```
poles(sys_cl)
```

```
ans (a column vector) =
-1 + 1 j
-1 - 1 j
-2
-4
-3 + 3 j
-3 - 3 j
```

You can choose to scale the system output here for zero steady-state error in the step response. This is accomplished in an intuitive manner, dividing the system `sys_cl` by the desired scaling factor.

```
sys_cl = sys_cl/51.76;
v_obc = step(sys_cl, 0:.1:10);

plot (v_obc, {xlab = "Time", ylab = "Magnitude"})
```

In Figure 1-9 the step response shows zero-steady-state error, little overshoot, and a response time of less than seven seconds.

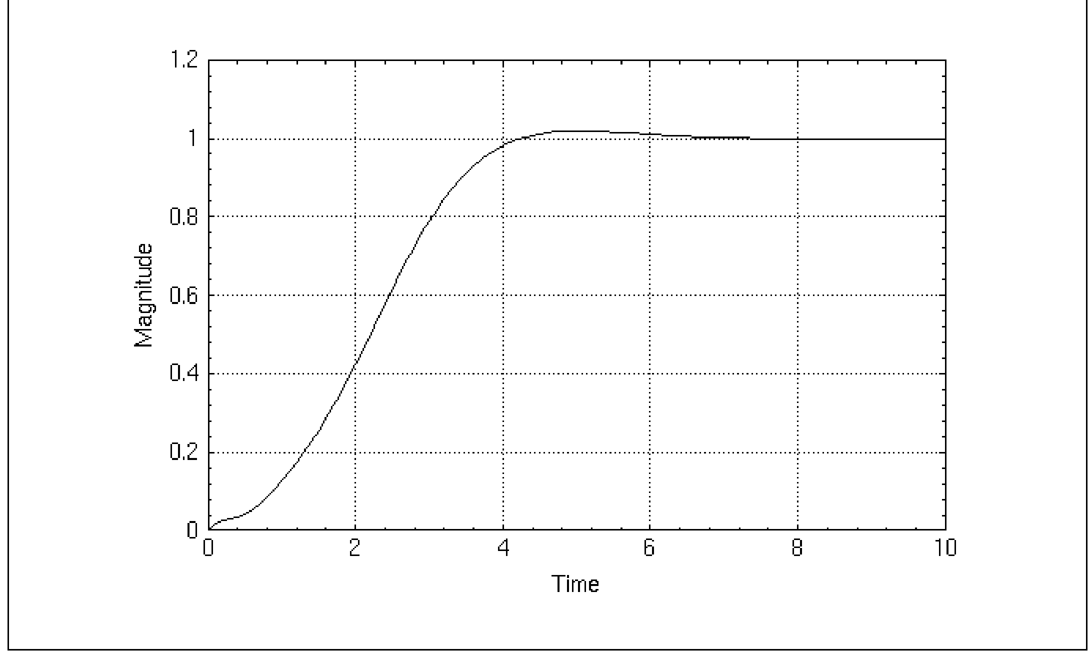


Figure 1-9. Step Response for Observer-Based Design

The system response is quite good, implying that your state estimates were satisfactory. You can do some further simulation, this time returning all the states directly from the original plant, and get a graphical picture of how the estimates track the actual states. First, you need to create the closed-loop system with all states available.

The `abcd( )` function extracts the A, B, C, and D matrices from a system object. When you call it here, all you are interested in is the closed-loop A matrix, so you do not need to extract the other state-space matrices.

```
A_cl = abcd(sys_cl);
```

When you create the estimator system `sys_est`, you use the original A matrix for the state-update equation, but you provide a zero external input (a B matrix of zero). The output matrix is an identity matrix passing back the three real state values and the three estimated state values as output, again with no external input values affecting the output. Here you use the optional `system( )` keyword `X0` to set the real state values to [1,2,3] and the estimated state values to [–1,–2,–3].

By simulating with a general input over two seconds, you can see how long it takes for the state values provided by the estimator to correct the incorrect initial conditions and track the real state values.

```
[,,allStates]=names(sys_cl);
sys_est = system(A_cl, zeros(6,1), eye(6,6), ...
    zeros(6,1),{x0 = [1,2,3,-1,-2,-3], ...
    stateNames = allStates});
state_resp = sys_est*pdm(ones(100,1), 0:(2/99):2);
```

Plot the results, referring to Figure 1-10:

```
plot(state_resp,{strip=2,xlab="Time",
    legend=["State","State estimate"]})
```

Even in the relatively short time span of this simulation, the estimates and the real states quickly converge.

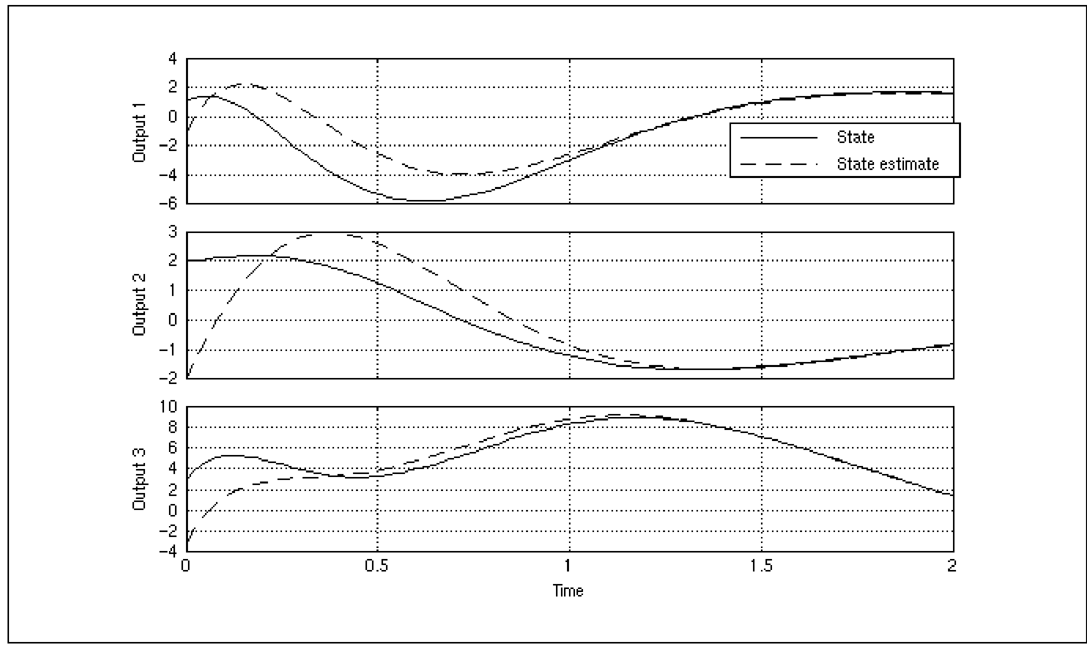


Figure 1-10. Multiple Plots Showing Time Needed for States to be Correctly Tracked by Estimator, Given Incorrect Initial Values

Helicopter Hover Problem: Discrete Formulation

Discrete-time control systems are most frequently designed in one of two ways: either directly implemented in the discrete domain, or first solved as continuous problems—often deriving directly from differential equations of motion—and then discretized. Here you take the second approach with the problem solved in the *Helicopter Hover Problem: State Feedback and Observer Design* section.

A guideline for choosing a sample rate for a system to be discretized is that it be significantly less than the smallest time constant of the continuous system divided by π.

Look at the open-loop pole magnitudes of your original open-loop continuous-time system `ssys`:

```
max(abs(ol_poles))
```

```
ans (a scalar) = 0.656513
```

You can use the default exponential discretization method with `dt` = 0.01 and compare frequency responses between the original system and the discretized system:

```
ssysd = discretize(ssys, 0.01);
f = freq(ssys,logspace(.001,10,200));
fd = freq(ssysd,logspace(.001,10,200));
```

In the following statements you compute the gain and phase of both systems and then plot them.

```
db = 20*log10(abs(f)); ph = (180/pi)*atan2(f);
dbd = 20*log10(abs(fd)); phd = (180/pi)*atan2(fd);
plot([db;ph;dbd;phd],{strip=2,xlog,
   ylab = ["Gain (dB)";"Phase (deg)"],
   x_lab = "Frequency (Hz)",
   legend = ["ssys";"ssysd"]})
```

In Figure 1-11 you can see the frequency responses match closely, indicating that this discretization method captures the continuous system's dynamics accurately.

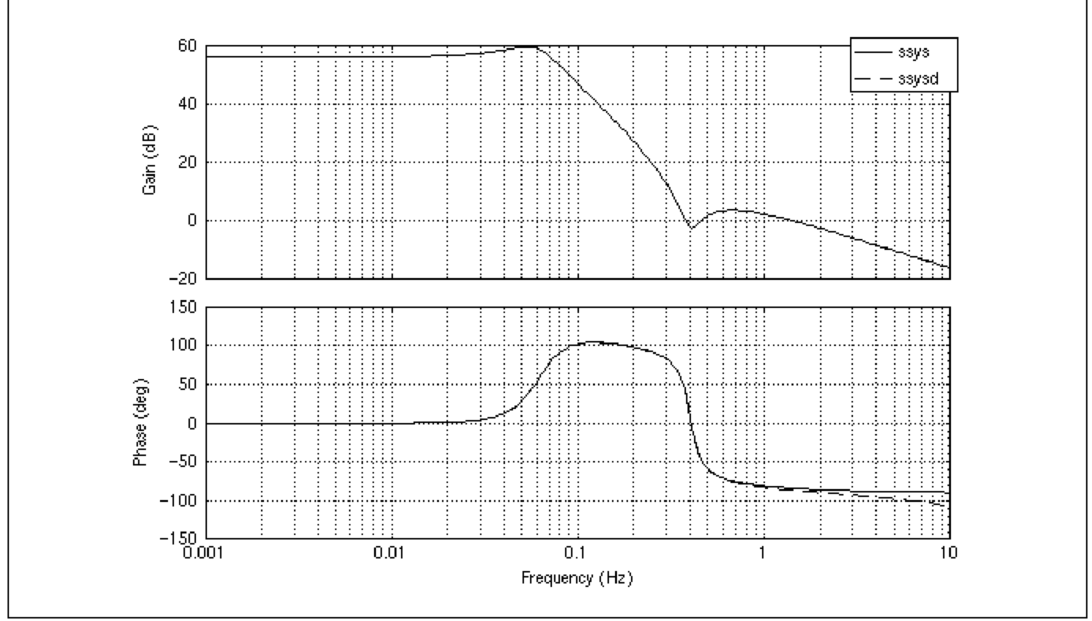


Figure 1-11. Frequency Response of ssys and Its Discrete Equivalent ssysd

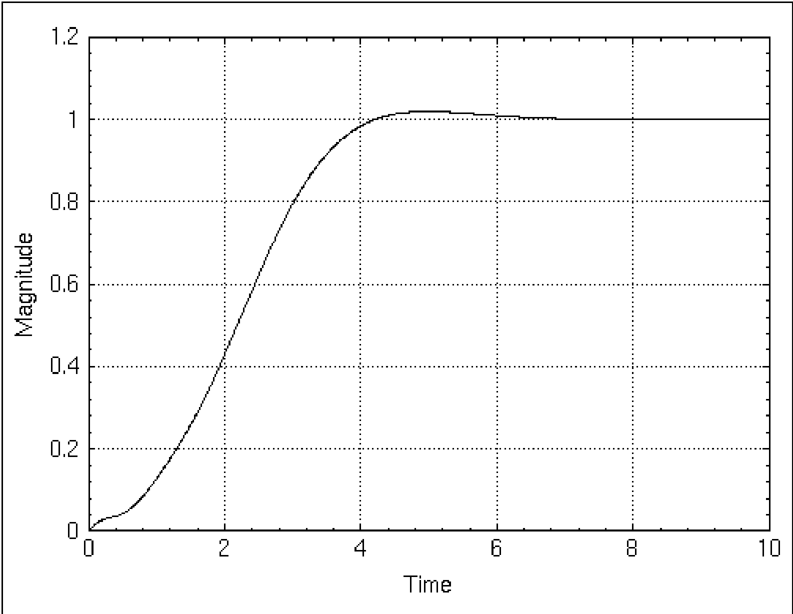


Figure 1-12. Step Response of a Discrete System Using Discretized Observer-Based Controller

As you discretize the compensator, form the closed-loop, scaled system, and simulate its response to a step input, you must ensure that the sampling interval is the same (`dt` = 0.01).

```
sys_obcd = discretize(sys_obc, 0.01);
```

```
sys_cld = feedback(ssysd,sys_obcd)/51.76;
v_cld = step(sys_cld, 0:0.01:10);
plot (v_cld, {xlab = "Time", ylab = "Magnitude"})
```

The resulting response is shown in Figure 1-12.

Inverted Wedge-Balancing Problem: LQG Control

[HW91] discusses an approach to balancing an inverted wedge by controlling the location of a sliding mass along the inside of the wedge. This example illustrates use of optimal control with a multi-input, multi-output (MIMO) system. This approach is based on minimizing a quadratic performance index with weight values based on the natural constraints of the system.

The linearized state-space equations, including the actuator and sensor dynamics, are as follows:

$$\begin{bmatrix} \dot{\theta} \\ \dot{x} \\ \ddot{\theta} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 15.54 & -10.93 & 0 & 0 \\ -5.31 & 0 & 0 & -16.24 \end{bmatrix} \begin{bmatrix} \theta \\ x \\ \dot{\theta} \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1.96 \end{bmatrix} u$$

$$y = \begin{bmatrix} 57.29 & 0 & 0 & 0 \\ 0 & 29.9 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ x \\ \dot{\theta} \\ \dot{x} \end{bmatrix}$$

θ is the angle (in radians) the wedge makes with the vertical axis, x is the position of the sliding mass, and u is the control input voltage. The outputs are scaled to give the measured angle in degrees and the measured position in meters.

```
A = [0,0,1,0;0,0,0,1;
    15.54,-10.93,0,0;
    -5.31,0,0,-16.24];
B = [0,0,0,1.96]';
C = [57.29,0,0,0;0,29.9,0,0];
D = [0;0];
```

```
states = ["Angle", "Mass Position",
"Angular Velocity","Mass Velocity"];
wsys = system(A,B,C,D,
    {inputNames= "Voltage",
    stateNames = states,
    outputNames=["Measured Angle","Measured
    Position"]});
```

You need to ensure that you have no uncontrollable or unobservable modes of the system:

```
[,,nuco] = minimal(wsys)
```

```
nuco (a scalar) = 0
```

Because there are no uncontrollable or unobservable states, you can proceed with the design of a regulator and estimator. The weighting matrix used here in designing the regulator reflects the desire to bring the value of the first state, the angle with the vertical, to zero as quickly as possible.

Because this system is open-loop unstable and has fairly fast poles in both halves of the s-plane, you want to make sure it can bring the effect of an external disturbance (such as a sharp push to the cart) to zero as quickly as possible.

```
[Kr,EVr,Pr] = regulator(wsys,diag([1e8,1,1,1]),1);
```

You then can verify that the regulator gain `Kr` can be used with full-state feedback to control this system by using an identity matrix for C to feed back the states:

```
[no, ni, ns] = size(wsys);
augwsys = system(A,B,eye(ns, ns),[]);
```

creating the compensator (which is a system object, though it has no states and thus has `NULL` A, B, and C matrices) with the gains `Kr`:

```
comp = system([],[],[],Kr);
```

and feeding back the states:

```
wsysreg = feedback(augwsys, comp);
```

You then can observe the system response to a sustained disturbance by simulating a five-second step response:

```
stepreg = step(wsysreg, 0:0.01:5);
plot (stepreg, {legend=states,
   xlab="Time",ylab="Magnitude",
   title="System Step Response with "+...
      "Full State Availability",!grid})
```

The resulting plot is shown in Figure 1-13.

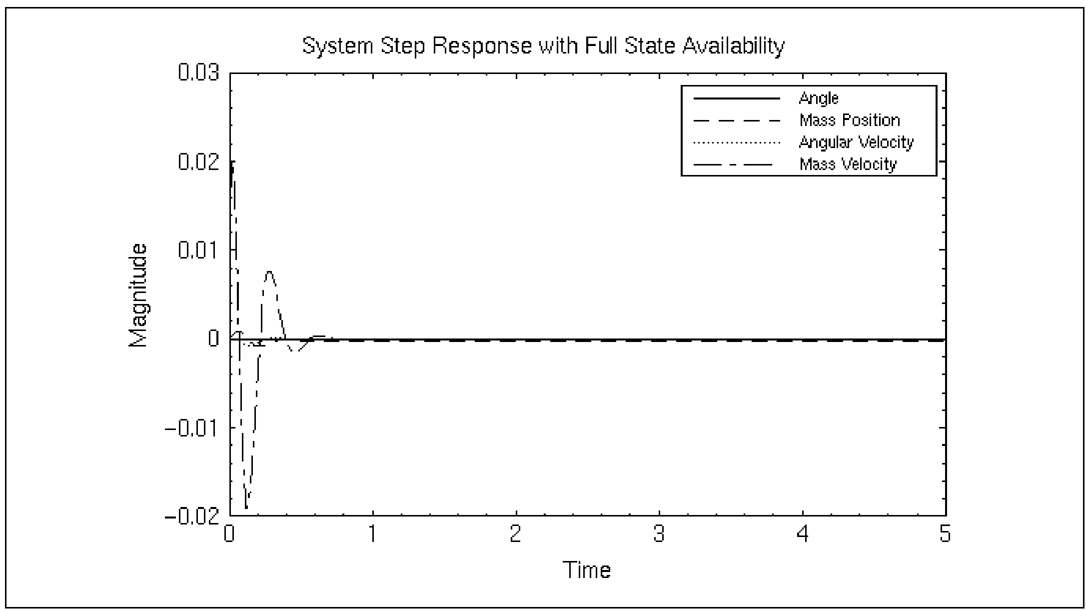


Figure 1-13. Response of Full-State Feedback Controller to a Unit Step Disturbance

Having established your regulator design, you build the estimator and simulate performance of the closed-loop system feeding back state estimates. You select the weights for the estimator based on the assumption that the state noise intensities corresponding to the wedge angle are smaller than those corresponding to the wedge position. The output weight matrix reflects your higher priority on the wedge angle than position.

The following steps generate the plot shown in Figure 1-14:

```
[Ke,EVe,Pe] = estimator(wsys,
   diag([1e-3,1,1e-3,10]), diag([14,0.01]));
wcomp = lqgcomp(wsys,Kr,Ke);
wlqg = feedback(wsys,wcomp);
resp = step(wlqg,0:0.01:3);
plot (resp, {legend = names(wlqg),
   xlab = "Time",ylab = "Magnitude",
   title = "Observer-Controller System "+...
           "Step Response",!grid})
```

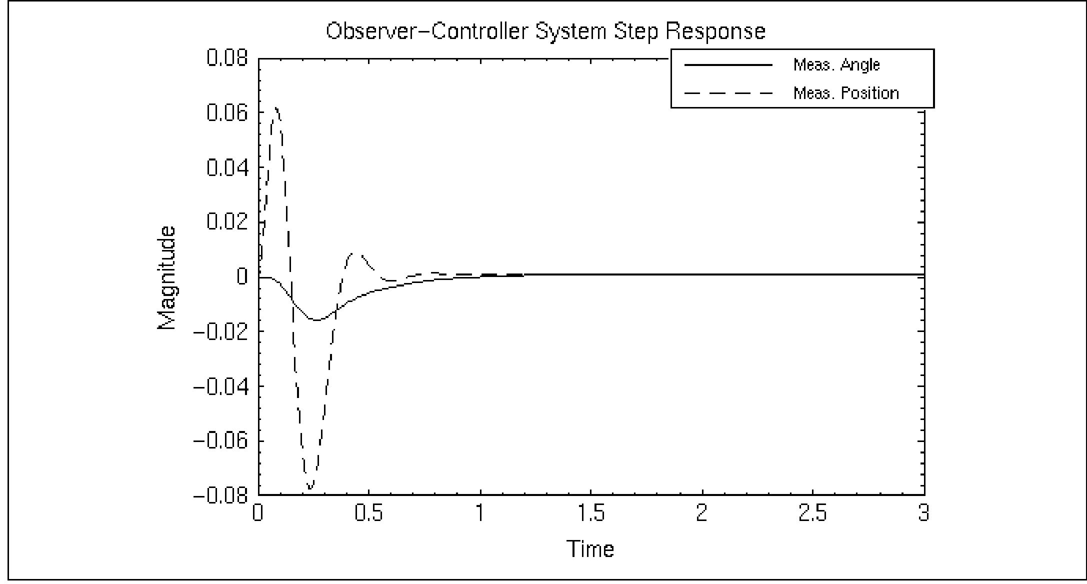


Figure 1-14. Response of Observer-Based Controller to a Unit Step Disturbance

2

Linear System Representation

Xmath provides a structure for system representation called a *system object*. This object includes system parameters in a data structure designed to reflect the way these systems are analyzed mathematically. Operations on these systems are likewise defined using operators that mirror as closely as possible the notation control engineers use. This chapter outlines the types of linear systems the system object represents and then discusses the implementation of a system within Xmath. The functions used to create a system object and to extract data from this object are an intrinsic part of the object and are also described. Finally, this chapter discusses the functions `check( )`, `discretize( )`, and `makecontinuous( )`, which use information stored in the system object to convert systems from one representation to another.

Linear Systems Represented in Xmath

Xmath handles finite-dimensional, linear, and time-invariant linear systems in both discrete and continuous time. These systems take one of the forms shown in Table 2-1.

Table 2-1. Summary of Linear Systems

System Type	Continuous Time	Discrete Time
State-spec	$\dot{x} = Ax + Bu$ $y = Cx + Du$	$x_{k+1} = Ax_k + Bu_k$ $y_k = Cx_k + Du_k$
Transfer function	$H(s) = C(sI - A)^{-1}B + D$	$H(z) = C(zI - A)^{-1}B + D$

The transfer function representation can be used to describe single-input, single output (SISO) systems only; there are no restrictions on the number of input and outputs that can be specified for a state-space system. All of these systems can be created using the Xmath `system( )` function.

Transfer Function System Models

One way of representing continuous-time finite-dimensional linear time-invariant systems is with the transfer function:

$$H(s) = \frac{num(s)}{den(s)}$$

with *num(s)* and *den(s)* being polynomials in *s*. They can be specified either by their roots or their coefficients. Transfer functions are defined using the Laplace transform operators for continuous time and the forward shift operator *z* for discrete time. Both forms of transfer functions are written with positive coefficients, each higher order terms having successively larger coefficients.

Discrete systems are defined analogously, using the *z* variable instead of *s*. Xmath does not automatically perform cancellations of polynomial roots appearing in both the numerator and the denominator of a transfer function. If you want to cancel common roots in a transfer function, use the function `cancel( )`. For state-space systems, refer to the `minimal( )` function. For more information, refer to the *Minimal Realizations* section of Chapter 6, *State-Space Design*.

To illustrate how you arrive at a particular transfer function, if you have a system differential equation that takes the form:

$$\ddot{y} + 6\dot{y} + 8y = 2\dot{u} - u \tag{2-1}$$

Laplace-transforming equation (assuming zero initial conditions) yields:

$$s^2 Y(s) + 6sY(s) + 8Y(s) = 2sU(s) - U(s) \tag{2-2}$$

Collecting terms, you can find the transfer function from $U(s)$ to $Y(s)$, $H(s)$:

$$\frac{Y(s)}{U(s)} = H(s) = \frac{2s - 1}{s^2 + 6s + 8} \tag{2-3}$$

The roots of the numerator polynomial are the zeros of the transfer function, and the roots of the denominator are its poles. In some circumstances, you might want to construct a transfer function based on where you know the pole and zero locations to be. For example, you can

form the same transfer function as that derived in the preceding transfer function equation using known pole, zero, and gain values:

$$H(s) = \frac{2(s-0.5)}{(s+2)(s+4)} \tag{2-4}$$

The systems represented in Equations 2-3 and 2-4 can be represented using Xmath's system objects, as shown in Example 2-1.

The Xmath transfer function system object currently can be used to represent single-input, single-output systems only. State-space form can be used to describe systems with multiple inputs or outputs. For more information, refer to the *State-Space System Models* section.

Example 2-1 Creating Transfer Functions

The polynomials in the numerator and denominator of the transfer function in Equation 2-3 are both in coefficients form, (described using just coefficients, not roots). `makepoly( )` creates two polynomials and passes them to the `system( )` function:

```
num3 = makepoly([2,-1],"s");
den3 = makepoly([1,6,8],"s");
H3 = system(num3,den3)
```

This displays as:

```
H3 (a transfer function) =
   2s - 1
   ----------
   s^2 + 6s + 8
initial integrator outputs
0
0
Input Names
-----------
Input 1
Output Names
------------
Output 1
System is continuous
```

The three statements used to create the transfer function could be more compactly combined as one. The use of *s* as the variable in which to express the transfer function is optional. Any variable, including the default *x*, can

be used so long as a consistent choice of variable is used for both numerator and denominator polynomials.

The transfer function in pole-zero-gain form from the preceding equation can be similarly implemented using the `polynomial( )` function to specify the numerator and denominator by their roots.

Note The `/` operator also can be used to create systems in transfer function form, as an alternative to using `system( )`.

```
H4 = 2*polynomial(0.5,"s")/polynomial([-2,-4],"s")
```

which displays as:

```
H4 (a transfer function) =

   2(s - 0.5)
   --------------
   (s + 2)(s + 4)

initial integrator outputs
0
0
Input Names
-----------

Input 1

Output Names
------------

Output 1

System is continuous
```

In both of these cases you have created a continuous system. Systems created in Xmath contain sample rate information as well as the numbers representing system dynamics. However, unless a sample rate is explicitly given as a keyword to `system( )`, it defaults to zero and the system is continuous. For an illustration of how to create a discrete system, refer to Example 2-2. The full discussion of the `system( )` function in the *system( )* section contains a listing of all the keywords associated with `system( )`.

State-Space System Models

State-space models comprise the second category of linear system representations in Xmath. In state-space form, first-order differential (continuous-time) and difference (discrete-time) equations are represented as a set of state and output updates. The states are represented by a vector *x*; *u* and *y* are vectors with as many elements as there are inputs and outputs, respectively. This system model is useful for representing multi-input, multi-output (MIMO) systems.

continuous time:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

discrete time:

$$x_{k+1} = Ax_k + Bu_k$$

$$y_k = Cx_k + Du_k$$

A straightforward mathematical transformation from the state-space form to the transfer function form is as follows:

$$H(q) = C(qI - A)^{-1}B + D$$

All of the forms represented in these equations can be represented using Xmath's system objects, as shown in Example 2-2.

Example 2-2 Creating a Discrete State-Space System

Suppose you have a system which you describe in state-space form as:

$$x_{k+1} = \begin{bmatrix} 0 & 1 \\ -0.75 & 0 \end{bmatrix} x_k + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_k$$

$$y_k = \begin{bmatrix} 0 & 1 \end{bmatrix} x_k$$

and you know that the sample period of the system is 0.5 seconds between samples—that is, the states and outputs are updated at every discrete interval *k*, consisting in this case of 0.5 seconds.

Again, you create the system using the `system( )` function. This time you use the optional `dt` keyword to indicate that this system is discrete.

```
A = [0,1;-0.75,0];
B = [1,0]';
C = [0,1];
D = 0;
sys4 = system(A,B,C,D, {dt = 0.5})
```

```
sys4 (a state space system) =
   A
      0    1
      -0.75 0
   B
      1
      0
   C
      0    1
   D
      0
   X0
      0
      0
System is discrete, sampling at 0.5 seconds.
```

Although five lines of MathScript were used to be as explicit as possible in creating this system, the call to `system( )` can encompass all of them.

Note When you create a system object, its inputs (A, B, C, D) are no longer needed.

Basic System Building Functions

The functions discussed in the following sections are available with the general Xmath package. However, Control Design Module users will find these functions an intrinsic part of their work, warranting this discussion. The *Xmath Help* provides additional details about these functions and examples of their use.

system( )

```
Sysd=system(A,B,C,D,{dt,inputNames,
   outputNames,stateNames,X0})
Sys = system(num,den,{dt,inputNames,outputNames})
Sys = system(Sys,{keywords})
```

The `system( )` function can create both the transfer-function and state-space forms of the system object. It requires four compatibly-sized matrices to create a state-space system, or a pair of polynomials to create a transfer function.

You can use optional keywords to store additional information about your system. Assigning `dt` to a positive scalar value indicates that the system is discrete, with a sampling period equal to that value. If `dt` is not specified, the system is continuous, with a sampling period defaulting to zero. Because information indicating whether the system is continuous or discrete is encapsulated within the system object itself, Xmath does not have separate functions for discrete- and continuous-time system analysis. Systems can be recognized by Xmath's functions as discrete or continuous using the `check( )` function and handled accordingly. For more information, refer to the *Using check( ) with System Objects* section. The capability to assign a discrete sample rate does not actually discretize a continuous-time system, however. For information on discretizing a system, refer to the *Discretizing a System* section.

A shortcut for creating state-space systems with an all-zero D matrix is to use a null-matrix specifier (`[]`) for the D matrix instead of entering an appropriately sized zero matrix. This will automatically set the D matrix to be a zero matrix with row size equal to the row size of C, and column size equal to the column size of B.

In addition, descriptive names for the inputs and outputs of a system can be specified as vectors of string names and assigned to the `inputNames`, `outputNames`, and `stateNames` keywords. `stateNames` is valid only when used in conjunction with a state-space system, as is the keyword `X0`, which can be used to set a vector of initial values for the states.

When you have created a system, you can modify it by changing the values of any of the keywords discussed in this section by calling `system( )` with the appropriate keyword setting.

Examples 2-1 and 2-2 illustrate how `system( )` can be called to create a transfer function and state-space system, respectively. `system( )` also can be used to change the attributes of an existing system.

Note In Example 2-3, the `[]` notation indicates that the D matrix should be an appropriately sized (in this case, scalar) zero matrix.

Example 2-3 Using system( ) to Change the Attributes of an Existing System

```
sys4=system([0,1;-0.75,0],[1,0]',[0,1],[],
    {dt=0.5});

sys4 = system(sys4, {inputNames = "Current",
    outputNames = "Velocity",
    stateNames = ["Torque","Angle"]})

sys4 (a state space system) =
    A
        0     1
        -0.75 0
    B
            1
        0
    C
        0     1
    D
        0
    X0
        0
        0

State Names
-----------
Torque Angle

Input Names
-----------
Current

Output Names
------------
Velocity

System is discrete, sampling at 0.5 seconds.
```

abcd( )

```
[A,B,C,D,X0] = abcd(Sys)
```

The `abcd( )` function extracts the component A, B, C, and D matrices described in equations from a state-space system object as shown in the *State-Space System Models* section. In addition, it returns the initial conditions on the states if a fifth output argument is requested.

`abcd( )` can be called on systems in either state-space or transfer function form. If the system is a transfer function, the conversion to state-space is

done internally to return A, B, C, and D, though the format of the variable `Sys` itself remains unchanged. The transfer function must be proper.

Using the systems defined in Examples 2-1 and 2-2, Example 2-4 illustrates the use of `abcd( )`.

Example 2-4 Using abcd( ) to Extract the State-Space Matrices

```
H3=makepoly([2,-1],"s")/makepoly([1,6,8],"s");
sys4=system([0,1;-0.75,0],[1,0]',[0,1],0,
    {dt=0.5});
```

You can extract the state-space matrices from each.

Note For the transfer function `H3`, an internal conversion is performed.

```
[A3,B3,C3,D3] = abcd(H3)?
A3 (a square matrix) =
    -2   1.58114
     0  -4
B3 (a column vector) =
     0
     2
C3 (a row vector) = -1.58114   1
D3 (a scalar) = 0
[A4,B4,C4,D4,X0] = abcd(sys4)
A4 (a square matrix) =
     0    1
    -0.75 0
B4 (a column vector) =
     1
     0
C4 (a row vector) = 0   1
D4 (a scalar) = 0
X0 (a column vector) =
     0
     0
```

numden( )

```
[num,den] = numden(Sys)
```

The `numden( )` function returns the numerator and denominator polynomials comprising a single-input, single-output system in transfer function form. If the system is in state-space form, an internal conversion is performed to find the transfer function equivalent, but the format of the system variable itself remains unchanged. State-space systems used in conjunction with `numden( )` must be single-input, single-output.

As noted in the *Transfer Function System Models* section, common roots in the numerator and denominator polynomials are not canceled.

Example 2-5 uses the state-space system from Example 2-2 to illustrate the use of `numden( )`.

Example 2-5 Using numden( ) to Extract the Transfer Function Polynomials

```
sys4=system([0,1;-0.75,0],[1,0]',[0,1],0,
    {dt=0.5});
[num,den] = numden(sys4)?
num (a polynomial) =
    -0.75
den (a polynomial) =
    (z^2 + 0.75)
```

Because `num` and `den` are polynomial objects and not a complete system, the discrete sampling time is not explicitly saved. You can use `check( )` with the `convert` keyword to map the two internal representations to each other, as described in the *Using check( ) with System Objects* section. However, notice that *z* was used as the polynomial variable, indicating that these numerator and denominator polynomials were obtained from a discrete-time system. Had the system been continuous, *s* would have been used instead of *z*.

period( )

```
dt = period(Sys)
```

The `period( )` function extracts the sample period (in seconds) of a system. If the system is continuous, `period( )` will return zero.

In Example 2-5, you found the numerator and denominator polynomials corresponding to the discrete state-space system. Example 2-6 combines

these polynomials into a transfer-function and uses `period( )` to set the sampling interval to match that of `sys4`.

Example 2-6 Using period( ) to Extract the Sampling Period

```
[num,den]=numden(sys4);

H4 = system(num,den,{dt = period(sys4)})

H4 (a transfer function) =
    -0.75
  -----------
  (z^2 + 0.75)

System is discrete, sampling at 0.5 seconds.
```

`check( )` provides a more concise means of converting between state-space and transfer function form, as described in the *Using check( ) with System Objects* section, but this example illustrates how the output of one function can be specified directly as keyword input to another.

names( )

```
[outputNames,inputNames,stateNames] = names(Sys)
```

The `names( )` function extracts matrices of strings representing the input, output, and (if the system is in state space form) state names of a system.

`names( )` also can be used to extract information from the PDM and polynomial objects. More information on these functions can be found in the *MATRIXx Help*.

When you create a system without specifying any names, a default set of names are assigned to it. Unlike user-specified names, these default names are not displayed in the **Xmath Commands** window. However, all, or any subset of the names you select to store with the system still can be extracted using `names( )` as shown in Example 2-7.

Example 2-7 Using names( ) to Extract the Variable Names Associated with a System

```
H3 = system(makepoly([2,-1],"s"),
   makepoly([1,6,8],"s"));
[outputNames, inputNames] = names(H3)

outputNames (a string) = Output 1
inputNames (a string) = Input 1

sys5=system([0,1;-0.75,0],[1,0]',[0,1],0,{dt=0.5,
   inputNames = "Current",
```

```
    outputNames = "Velocity",
    stateNames = ["Torque","Angle"]});
[,,stateNames] = names(sys5)?
statenames (a row vector of strings)=Torque Angle
```

Size and Indexing of Dynamic Systems

The size of a system object is defined by how many outputs, inputs, and (in the case of a state-space system) states it has. You can use the `size( )` function to find these dimensions.

You can index into a dynamic system to create a new dynamic system which has a subset of the original inputs and outputs:

$Sys = Sys1(i,j)$ is defined to be a system such that $y = y1(i)$ and $u = u1(j)$. i and j can both be vectors as well, in which case multiple inputs and outputs will be extracted.

The previous definition of indexing was designed with the traditional definition of a transfer function in mind.

$$y(q) = Sys(q) \times u(q)$$

Using check( ) with System Objects

Several common attributes of systems can be easily determined using Xmath's ability to distinguish between object types and characteristics. You can use the `check( )` function with systems, as shown in Example 2-8, to determine whether a system is in transfer function or state-space form, discrete, continuous, or stable. In addition, you can use `check( )` with the `convert` keyword to change a system's representation between SISO state-space and transfer-function forms.

Example 2-8 Using check( ) with a System

```
a = [1.875,0;0,-0.26];
b = [1;0];
c = [0.5,1];
d = 0;
sys = system(a,b,c,d, {dt = 0.001});
```

Because this system is discrete and has a pole where magnitude exceeds 1, it is not stable.

```
check(sys, {stable})
ans (a scalar) = 0
check(sys, {discrete, ss})
ans (a scalar) = 1
[, tfsys] = check(sys, {tf, convert})
tfsys (a transfer function) =
   (z + 0.26)
---------------------
(z + 0.26)(z - 1.875)
initial delay outputs
0
0
System is discrete, sampling at 0.001 seconds.
```

Discretizing a System

Many systems where behavior derives from physical equations of motion can be modeled most naturally as continuous processes, using differential equations. Therefore, you often choose to discretize these models for use with a digital controller. A number of mathematical methods have been developed to approximate the behavior of a continuous system in a discrete-time representation with an appropriately fast sampling rate. Xmath provides two functions, `discretize( )` and `makecontinuous( )`, which encompass a range of these techniques. `discretize( )` converts a system from its representation as a continuous function in the s-domain to a discrete-time z-domain function. `makecontinuous( )` does the reverse, transforming a discrete system to its continuous form.

discretize( )

```
SysD = discretize(Sys,{dt,exponential,forward,backward,
   tustins,ztransform,polezero,firstorder})
```

The `discretize( )` function has a number of keywords that correspond to the different methods of continuous-to-discrete conversion that are implemented within Xmath. The sampling interval (in seconds) for the discrete system should be set equal to the keyword `dt`. If no value for `dt` is specified, a default of 0.5 seconds is used. The default discretization method used is the exponential (step-invariant) transform. The different

discretization methods used based on the specification of each keyword are discussed in the following sections.

Numerical Integration Methods: forward, backward, tustins

Xmath provides three methods of numerical integration of a differential transfer function: the forward and backward rectangular rules, and *Tustin's* rule (also called the *bilinear* or *trapezoidal* transform).

To convert the system description from a continuous differential equation to a discrete difference equation, you approximate the value of the derivative in the continuous equation over each `dt` seconds of time, then find the area of the geometric region having width `dt` and height equal to the derivative. You can do this in a number of ways, as discussed in [FPW90].

For the forward rectangular method, you assume the incremental area term between sampling times $k * dt$ and $(k + 1) * dt$ to be a rectangle having width dt and height equal to the integral form of the differential equation at time $(k + 1) * dt$. In essence, you get your amplitude estimate for each rectangle by looking forward, hence the name. The backward rectangular method arises similarly, except that you get the rectangle's height by looking backward and taking the value of the integral at $k * dt$. The forward rectangular approach tends to overestimate the incremental area somewhat and the backward approach tends to underestimate it (though with a sufficiently small sampling interval, this may not pose a large problem). The trapezoid rule strikes a balance between these two methods by taking the average of the rectangles defined by the forward and backward methods and using that value as the incremental area in approximating the difference equation.

These approaches can be summarized as substitutions between the continuous-time Laplace-transform operators and the discrete z-transform operator z as shown in Table 2-1.

Table 2-2. Mapping Methods for discretize( )

Method of Approximation	Continuous to Discrete
Forward rectangular rule: Keyword: `forward`	$s \rightarrow \frac{z-1}{dt}$

Table 2-2. Mapping Methods for discretize( ) (Continued)

Method of Approximation	Continuous to Discrete
Backward rectangular rule: Keyword: `backward`	$s \to \frac{z-1}{zdt}$
Tustin's rule: Keyword: `tustins`	$s \to \frac{2(z-1)}{dt(z+1)}$

Pole-Zero Matching: polezero

The pole-zero matching method of discretizing a continuous system follows from the relation between the continuous s and discrete z frequency domains:

$$z = e^{sT}$$

where T is the sampling interval to be used for the discrete system. Continuous-time poles and finite zeros are mapped to the z-plane using this relation. Zeros at infinity are mapped into $z = 0$, where they do not affect the frequency response.

After the poles and zeros have been mapped, the algorithm tries to make sure the system gains are equivalent at some critical frequency. If the systems have no poles or zeros at $DC(s = 0, z = 1)$, the discrete-time gain is selected such that the system gains match at DC. Alternatively, if the systems have no poles or zeros at the Nyquist frequency ($s = p * \mathrm{j}/T$, $z = -1$), the gains are equalized at that frequency. In the event that neither of these gains can be matched, no gain is chosen.

Z-Transform: ztransform

This method is a direct Z-transform of the continuous-time transfer function, which corresponds to the Z-transform of the impulse response of the system. If `ztransform` is used, you will match the impulse responses of the continuous and discrete systems. The responses may differ slightly due to round off error.

Hold Equivalence Methods: exponential and firstorder

The discretization methods for `exponential` and `firstorder` both rely on the approximation that the discrete-time response can be represented as a hold on the sampled values of the continuous-time response.

The `exponential` keyword assumes that the response value between samples is constant and can, therefore, be represented by a zero-order hold polynomial. When `exponential` is specified, the continuous-time step response is discretized using the Z-transform, then the result is divided by the Z-transform of a step $z/(z-1)$ to produce the desired transfer function.

The `firstorder` keyword assumes extrapolation between samples (connecting sample to sample in a straight line). If `firstorder` is specified, the continuous-time ramp response is discretized using the Z-transform and then the result is divided by the Z-transform of a ramp $z * dt/(z-1)2$ to produce the desired transfer function.

In each of these cases, the appropriate response (impulse, step, or ramp) will match the continuous response very closely, with the only error being round off error.

While no one method of discretization will always perform best for all systems and all sampling times, it is often a good idea to compare the frequency response resulting from different discretized models to the continuous response. Example 2-9 applies the forward, backward, tustins, exponential, and matched pole-zero discretization methods.

Example 2-9 A Comparison of Several Discretization Methods

```
H = system(0.5*polynomial([-0.36]),
    makepoly([1,2.79,2.74,1.11,0.16]));
```

Create a logspaced vector for the frequency range of the response:

```
F = logspace(.001,5,200);
```

Perform the discretization using the different algorithms:

```
Hd_f = discretize(H,0.1,{forward});
Hd_b = discretize(H,0.1,{backward});
Hd_t = discretize(H,0.1,{tustins});
Hd_z = discretize(H,0.1,{polezero});
Hd_e = discretize(H,0.1,{exponential});
```

Now you can calculate the magnitude response as a function of frequency,

```
gainc  = 20*log10(abs(freq(H,F)));
gain_f = 20*log10(abs(freq(Hd_f,F)));
gain_b = 20*log10(abs(freq(Hd_b,F)));
gain_t = 20*log10(abs(freq(Hd_t,F)));
```

```
gain_z = 20*log10(abs(freq(Hd_z,F)));
gain_e = 20*log10(abs(freq(Hd_e,F)));
```

and plot it (as shown in Figure 2-1).

```
plot ([gainc,gain_f,gain_b,gain_t,gain_z,gain_e],
    {legend = ["Continuous", "Forward",
                  "Backward", "Tustins", "Pole Zero",
                  "Exponential"], x_log,
    xlab="Frequency (Hz)",ylab="Magnitude (dB)"})
```

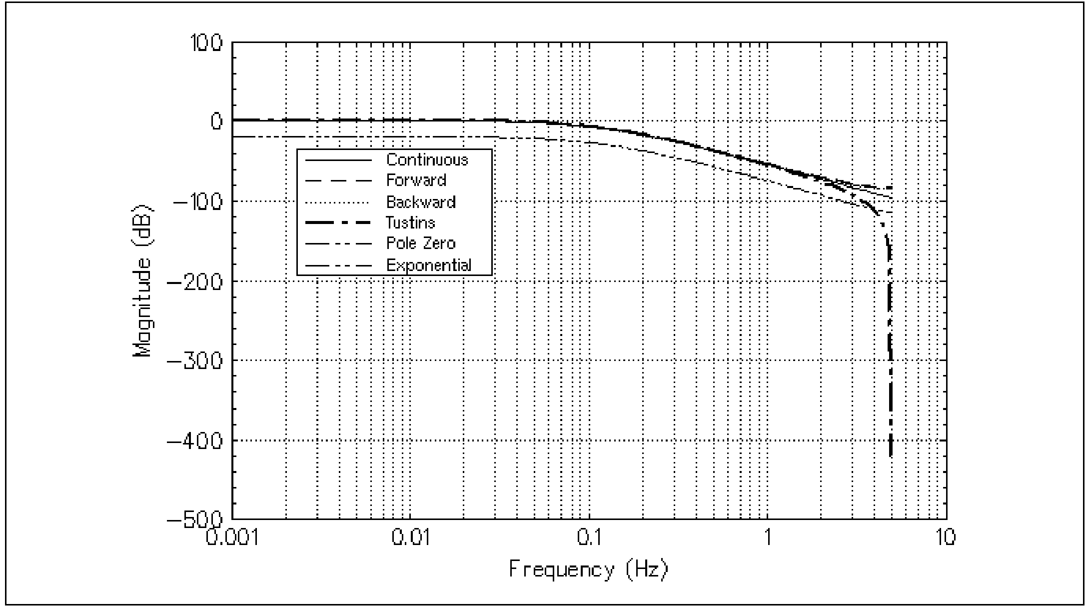


Figure 2-1. Comparison of Different Frequency Response Techniques

Although most of the discretizations used would give acceptable approximations to the continuous-time response, notice that most of them diverge greatly at higher frequencies. You may find it illustrative to run this example with larger and smaller sampling intervals to see how the choice of sampling rate, as well as the choice of method, affects the accuracy of the discretized frequency response.

makecontinuous( )

```
Sys=makecontinuous(SysD,{exponential, forward,
    backward,tustins, ztransform})
```

Many of the discretization techniques discussed in the *Hold Equivalence Methods: exponential and firstorder* section can be easily reversed to obtain a continuous equivalent to a discrete system. The `makecontinuous( )` function implements these reverse algorithms based on the keyword you specify as shown in Example 2-10. Although `makecontinuous( )` accepts an input system in any form, it returns the continuous-time system as a state-space system.

The forward, backward, and Tustin methods for mapping from the s-plane to the z-plane can be easily reversed using the equivalencies shown in Table 2-3.

Table 2-3. Mapping Methods for makecontinuous( )

Method of Approximation	Discrete to Continuous
Forward rectangular rule: Keyword: `forward`	$z \to 1 + s(dt)$
Backward rectangular rule: Keyword: `backward`	$z \to \frac{1}{1 - s(dt)}$
Tustin's rule: Keyword: `tustins`	$z \to \frac{1 + s(dt)/2}{1 - s(dt)/2}$

Discrete-to-continuous algorithms using matrix logarithms (to reverse the exponential operations involved in doing the z-transform for the impulse invariant zero-order hold) are available for the `exponential` (step-invariant) transformation and the `ztransform` (impulse-invariant) methods. A limitation of these methods, however, is that they will not return a meaningful continuous equivalent to a discrete system that has pure delays ($1/z$ terms), because the logarithm of zero is infinite.

Example 2-10 Verifying a Discretization Using makecontinuous( )

Create a system:

```
H = 0.5*polynomial([-0.36])/...
   polynomial([-1,-1,-0.395+0.06305*jay,
   -0.395-0.06305*jay]);
```

Form the discrete equivalent using the forward approximation:

```
Hd_f = discretize(H,0.1, {forward});
```

Now convert back to the continuous form:

```
Hc = makecontinuous(Hd_f, {forward});
[num,den] = numden(Hc)

num (a polynomial) =

(s + 0.36)

den (a polynomial) =

                         2
(s + 0.999998)(s + 1)(s + 0.79s + 0.16)
```

Although `makecontinuous( )` restores the continuous-time poles and zeros, it cannot match gains precisely.

3

Building System Connections

Large system models are frequently built by connecting smaller models together. You can perform different types of linear system interconnections using the Xmath functions discussed in this chapter.

MathScript allows operators (*,+, and so on) to be *overloaded*—given different behaviors when used with different objects. A number of simple types of connections have been implemented as overloaded operators on systems, while more complex connections are available through specialized functions.

Linear System Interconnection Operators

Overloaded operators provide a quick way to perform different types of basic connections between systems. Table 3-1 illustrates these operations on a pair of systems Sys_1 and Sys_2 with outputs y_1 and y_2 and inputs u_1 and u_2, respectively.

Table 3-1. Summary of Interconnection Operators

Diagram	Description
`Sys = Sys1 + Sys2` u1, u, u2, Sys1, Sys2, y1, y, +, y2	Parallel connection where $y = y_1 + y_2$. The inputs are tied together where $u = u_1 = u_2$.
`Sys = Sys1 - Sys2` u1, u, u2, Sys1, Sys2, y1, y, –, y2	Parallel connection where $y = y_1 + y_2$. In the unary case, $Sys = -Sys_1$ where $y = -y_1$. The inputs are tied together where $u = u_1 = u_2$.

Table 3-1. Summary of Interconnection Operators (Continued)

Diagram	Description
`Sys = Sys2 * Sys1`	Cascade connection of Sys_1 and Sys_2 where the output of `Sys` is y_2 and the input is u_1.
`Sys = Sys1/Sys2`	Cascade connection of `Sys1` and inverted `Sys2` where $Sys = Sys_1 * inv(Sys_2)$, $u = u_2$, and $y = y_1$.
`Sys = Sys1\Sys2`	Cascade connection of inverted Sys_1 and Sys_2 where $Sys = inv(Sys_1) * Sys_2$, $u = u_2$, and $y = y_1$.
`Sys = [Sys1;Sys2`	Parallel connection where $y = [y_1;y_2]$. The inputs are tied together where $u = u_1 = u$.
`Sys = [Sys1,Sys2`	Parallel connection where $y = y_1 + y_2$ and $u = [u_1;u_2]$.
`Sys = Sys1'`	If Sys_1 is in state-space form and comprises the matrices (A_1,B_1,C_1,D_1), `Sys` comprises (A_1',C_1',B_1',D_1'). If Sys_1 is a transfer function, it is converted internally to state-space form.

Table 3-1. Summary of Interconnection Operators (Continued)

Diagram	Description
`Sys = adj[Sys1]`	If Sys_1 is in state-space form and comprises the matrices (A_1, B_1, C_1, D_1), `Sys` is the adjoint system and comprises ($-A_1'$, C_1', B_1', D_1'). If Sys_1 is a transfer function, it is converted internally to state-space form.
`p1/p2`	Alternate method to create a system, where `p1` and `p2` are the numerator and denominator polynomials, respectively; does not allow the use of keywords.

`Sys = inv(Sys1)`	The inverse (pseudoinverse) of a system can be found using $inv(Sys_1)$. If Sys_1 is a transfer function, $inv(Sys_1)$ is the reciprocal of the transfer function. If Sys_1 is a state-space system (`A1,B1,C1,D1`), then `Sys = system(A,B,C,D)` where `A,B,C,D` are defined as follows:
	`D = pinv(D1)`
	`A = A1-B1*D*C1`
	`B = B1*D`
	`C = -D*C1`

Dynamic systems also can be flexibly combined with scalars and compatibly sized matrices using the operators in Table 3-1. A compatibly sized matrix is one having the same dimensions as the dynamic system's D matrix (row size equal to the number of outputs and column size equal to the number of inputs).

Operations performed with a dynamic system and a matrix M as the operands internally handle M as a pure-gain system implemented as `system([],[],[],M)`.

The `*` operator can be used with a system and a PDM to find the time response of the system to the general input data stored in the PDM. For a detailed description of time simulation in Xmath, refer to the *General Time-Domain Simulation* section of Chapter 4, *System Analysis*.

Linear System Interconnection Functions

`afeedback( )`, `append( )`, `connect( )`, and `feedback( )` connect dynamic systems in state–space or transfer–function form to produce a larger system in state-space form. The following restrictions apply to all of these functions:

- Both systems must have the same sample rate.
- Improper dynamic systems (systems with more zeros than poles) are not allowed.
- If the systems to be connected are in transfer-function form, they must be expressed in the same dependent variable.

In describing the algorithms used in these connection functions, we will often refer to the component matrices of a state-space system Sys_1 as A_1, B_1, C_1, and D_1.

afeedback( )

```
Sys = afeedback(Sys1,{Sys2})
```

The `afeedback( )` function connects two dynamic systems in a feedback loop, and obtains a single system representation for the complex loop. `Sys` is organized as shown in Figure 3-1. Additional external inputs to the feedback path are included with outputs from the feedforward path. Figure 3-1 illustrates that outputs of the feedback path system are included with forward path outputs. For an example of how to use `afeedback( )`, refer to Example 3-1.

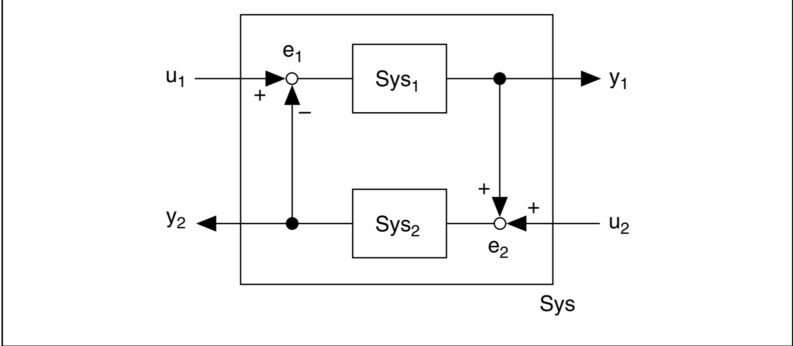


Figure 3-1. afeedback System Configuration

- By default, feedback is defined to be negative.
- The number of outputs from the first system must equal the number of inputs to the second system.
- The number of outputs from the second system must equal the number of inputs in the first.
- Both systems must have the same sample rate.
- Improper dynamic systems (systems with more zeros than poles) are not allowed.
- When only one system is specified, it must be square (it must have an equal number of inputs and outputs).

Example 3-1 Using afeedback( ) to Connect Two Systems

```
Sys1 = system([.5,1;0,2],[1,0]',[0,1],0);
Sys2 = system([1,-.2;1,0],[1,0]',[1,1],0);
saf = afeedback(Sys1,Sys2);
```

Algorithm

If only one system input (Sys_1) is provided to `afeedback( )`, the second input (Sys_2) defaults to a zero-state system with unity gain. This is analogous to a state-space system with `NULL` values for the A, B, and C matrices, and with an identity matrix for D. Notice that you use the Xmath definition of a non-square identity matrix. In this case, the row dimension of D equals the number of inputs to Sys_1, and the column dimension equals the number of outputs of Sys_1. In the following discussion, you denote the state-space matrices of Sys_1 by A_1, B_1, C_1, and D_1, and you follow the same convention for Sys_2.

The two systems are first internally converted to a state-space form, if necessary, and subdivided into the A, B, C, and D state-space matrices. Scaling matrices S1 and S2 are computed for Sys_1 and Sys_2 as follows:

$$S_1 = I + D_1D_2$$
$$S_2 = I + D_2D_1$$

Additionally, you define:

$$B_{1s} = B_1/S_2 \text{ and } D_{1s} = D_1/S_2$$
$$B_{2s} = B_2/S_1 \text{ and } D_{2s} = D_2/S_1$$

Matrix right-division problems must be well-posed, with the scaling matrices S_1 and S_2 nonsingular. `afeedback( )` displays an error message

if the condition estimate for either matrix is less than `eps`. For more information on this condition estimate, refer to the *MATRIXx Help* for the Xmath function `rcond( )`.

Using `Sys` to denote the state-space representation for the complete feedback loop, you can express its component matrices A, B, C, and D as combinations of the component matrices you obtained from Sys_1 and Sys_2. The full matrices used with two input systems are shown in the next example. In the case of a constant-gain feedback, A, B, C, and D are computed using only the matrix partitions shown in bold type.

The initial conditions for the systems are appended to each other columnwise.

$$A = \begin{bmatrix} A_1 - B_{1s}D_2C_1 & -B_{1s}C_2 \\ B_{2s}C_1 & A_2 - B_{2s}D_1C_2 \end{bmatrix}$$

$$B = \begin{bmatrix} B_{1s} & -B_{1s}D_2 \\ B_{2s}D_1 & B_{2s} \end{bmatrix}$$

$$C = \begin{bmatrix} S_1/C_1 & -D_{1s}C_2 \\ D_{2s}C_1 & S_2C_2 \end{bmatrix}$$

$$D = \begin{bmatrix} D_{1s} & -D_1D_{2s} \\ D_2D_{1s} & D_{2s} \end{bmatrix}$$

`afeedback( )` cannot be used with improper transfer functions—systems having more zeros than poles—because this algorithm is strictly state-space.

append( )

```
Sys = append(Sys1,Sys2)
```

The `append( )` function appends two dynamic systems in a form suitable for use with the `connect( )` function (refer to the *connect( )* section). `Sys`

is created by appending the inputs, outputs, and states of Sys1 and Sys2. A larger number of systems can be appended by appending two at a time.

- Both systems must have the same sample rate.
- Improper dynamic systems—systems with more zeros than poles—are not allowed, because Sys is represented in state-space form.

The output is a dynamic system in block form as shown in Figure 3-2.

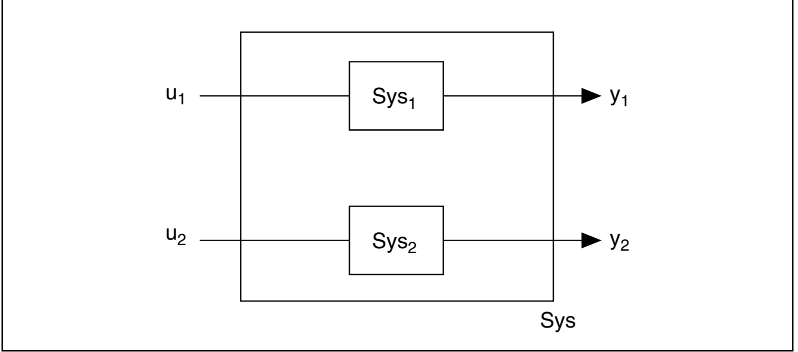


Figure 3-2. Output of a Dynamic System

For an example of how to use append( ), refer to Example 3-2.

Example 3-2 Using append( )

```
s3 = system(makepoly([1,2]), makepoly([1,3,5,0]));
s4 = system(makepoly([1]), makepoly([1,4,4]));
sap = append(s3,s4);
```

In the following discussion, the component state-space matrices of Sys_1 are denoted by A_1, B_1, C_1, and D_1, and you follow the same convention for Sys_2.

The algorithm for append( ) is done strictly using the state-space representations of Sys_1 and Sys_2. For this reason, Sys_1 and Sys_2 cannot be improper transfer functions (transfer functions having more zeros than poles). The component A, B, C, and D matrices of Sys_1 and Sys_2 are extracted using the abcd( ) function. The A, B, C, and D matrices comprising Sys are obtained as shown in the following example, where the zero matrix elements span as many rows and columns as necessary.

$$A = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} \qquad B = \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}$$

$$C = \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix} \qquad D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}$$

`append( )` performs a check to make sure both $\mathtt{Sys}_1$ and $\mathtt{Sys}_2$ have the same sample rate, and adopts this rate for the appended system. Any initial conditions on the states are also appended columnwise.

connect( )

```
Sys = connect(Sys1,{K,M,N})
```

The `connect( )` function performs a general interconnection around a system. This provides two basic capabilities:

- constant gain feedback
- general input–output interconnection

In its simplest form, `connect( )` can be used to wrap constant gain feedback around a system. The keyword `K`, used to specify feedback gain, also can be used to specify which outputs are fed back to the input of the system. By specifying the optional keywords `M` and `N`, you also can specify input and output gains.

General input–output interconnection is applicable to the block form system provided by `append( )`, as described in the *append( )* section. Parameters used in the `connect` command are illustrated in Figure 3-3. Notice that feedback is defined with a positive sign.

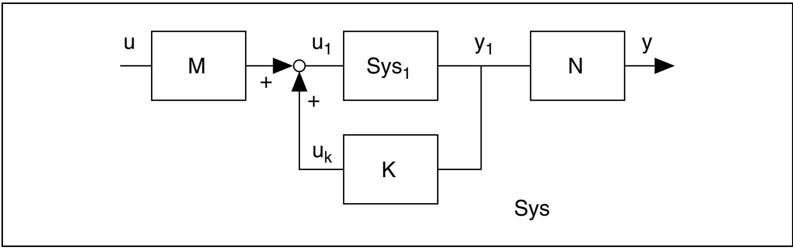


Figure 3-3. Parameters Used with the connect Command

- By default, feedback is defined to be positive. To enforce negative feedback, specify `connect(Sys,-K)`.
- A "selection matrix" has a single 1 in each row; the rest of the row contains zeros. This is useful for indicating the subset of system inputs and outputs to be used. In many cases, however, it is simpler to extract desired inputs and outputs through indexing.
- Both systems must have the same sample rate.
- Improper dynamic systems—systems with more zeros than poles—are not allowed.

The number of outputs in the combined system is the sum of the number of outputs from the two systems you are appending. For an example of how to use gains for the input, the output, and the fed-back data, refer to Example 3-3.

Example 3-3 Using connect( ) to Perform a General Output-Input Connection

```
tfsys=system(makepoly([1,2]),makepoly([1,3,0]))

tfsys (a transfer function) =

x + 2
-------
  2
x + 3x

initial integrator outputs
0
0

Input Names
-----------

Input 1
Output Names
------------
Output 1

System is continuous

connect(tfsys,0.12,2,1.5)

ans (a state space system) =

A
-3 1
-0.12 0.12

B
0
```

```
2

C
-1.5 1.5

D
0

X0
0
0
```

Algorithm

For the feedback system shown in Example 3-3, you can write the following system equations:

$$\dot{x} = A_1 x + B_1 u_1 \qquad y_1 = C_1 x + D_1 u_1$$

Combining these equations with the equation for the positive feedback input term:

$$u_1 = K y_1 + M u$$

and multiplying by the input and output gains *M* and *N*, you obtain the following state-space equations describing the entire system between input *u* and output *y*. If you do not specify any values for the gain matrices, *K* defaults to zero (no feedback) and *M* and *N* default to appropriately-sized identity matrices (unity gain on the input and output).

$$\dot{x} = (A_1 + B_1(I - KD_1)^{-1} KC_1)x + B_1(I - KD_1)^{-1} Mu$$
$$y = N(I - KD_1)^{-1} C_1 x + ND_1(I - KD_1)^{-1} Mu$$

This algorithm assumes that the closed-loop system is well posed to ensure that `Sys` will be proper. The $(I - KD_1)$ term must be invertible, and a warning appears if the condition estimate of the term (refer to `rcond`) is less than `eps`.

feedback( )

```
Sys = feedback(Sys1,{Sys2})
```

The `feedback( )` function connects two dynamic systems together in a feedback loop as shown in Figure 3-4.

- By default, feedback is defined to be negative.
- Both systems must have the same sample rate.
- Improper dynamic systems (systems with more zeros than poles) are not allowed.
- When only one system is specified, it must be square (it must have an equal number of inputs and outputs).

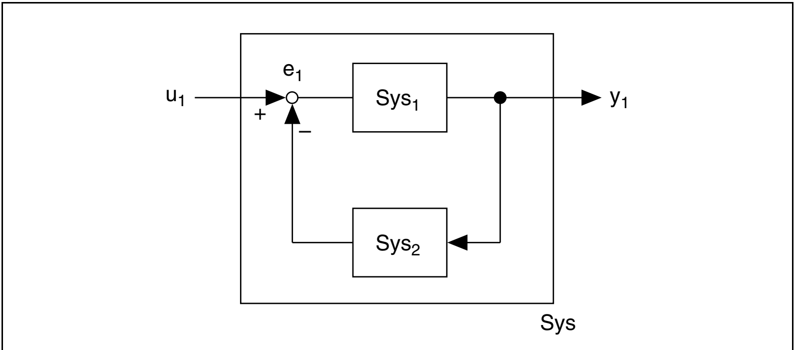


Figure 3-4. Feedback System Configuration

For an example of how to implement unity gain feedback using the `feedback( )` function, refer to Example 3-4.

Example 3-4 Implementing Unity Gain Feedback Using feedback( )

```
tfsys2 = polynomial(-1)/polynomial([-2,-3]);
feedback(tfsys2)  # Note that this implements
                  # negative unity gain feedback

ans (a state space system) =

A
-2 1
1 -4

B
0
1

C
```

```
-1 1

D
0

X0
0
0

State Names
-----------


Input Names
-----------
Input 1

Output Names
------------
Output 1

System is continuous
```

Algorithm

The system used for the feedback loop, `Sys2`, is optional. If it is not specified, a default state-space system is used with `NULL` matrices for A_2, B_2, and C_2 and an identity matrix for D_2 so that unity gain feedback is implemented.

From Figure 3-4 and the state-space definitions of the systems, you derive the following equations:

$$\begin{aligned}
\dot{x}_1 &= A_1x_1 + Be_1 \\
y_1 &= C_1x_1 + D_1e_1 \\
\dot{x}_2 &= A_2x_2 + B_2y_1 \\
y_2 &= C_2x_2 + D_2y_1 \\
e_1 &= u_1 - y_2
\end{aligned}$$

The single system resulting from the feedback combination of `Sys1` and `Sys2` has u_1 as its input, y_1 as its output, and a state vector consisting of the appended states of `Sys1` and `Sys2`. Using these five equations to find the state-space dynamics of the complete system results in the overall system description.

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \bullet$$

$$= \begin{bmatrix} A_1 - B_1(I + D_2D_1)^{-1}D_2C_1 & -B_1(I + D_2D_1)^{-1}C_2 \\ B_2(I + D_1D_2)^{-1}C_1 & A_2 - B_2(I + D_1D_2)^{-1}D_1C_2 \end{bmatrix} \bullet$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1(I + D_2D_1)^{-1} \\ B_2(I + D_1D_2)^{-1}D_1 \end{bmatrix} u_1$$

$$y_1 = \begin{bmatrix} (I + D_1D_2)^{-1}C_1 & -D_1(I + D_2D_1)^{-1}C_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} +$$

$$D_1(I + D_2D_1)^{-1}u_1$$

This algorithm assumes that the closed-loop system is well constructed (the $(I + D_2D_1)$ and $(I + D_1D_2)$ terms must be invertible). This condition ensures that the output system `Sys` will be proper.

4

System Analysis

This chapter discusses time-domain solutions of the equations underlying transfer functions and state-space system models, and what these solutions tell us about the stability of the system. Xmath provides a number of functions for performing this system analysis and computing the time-domain system response to both general and specific "standard" inputs.

Time-Domain Solution of System Equations

Given the state-space equations:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

you obtain:

$$x(t) = e^{At}x_0 + \int_0^t e^{A\tau}Bu(t-\tau)d\tau$$

letting x_0 denote any initial conditions on the system states. The integral term in the preceding equation defines a convolution integral. Using * to represent the convolution operator, the time-domain system output for all time $t \geq 0$ is:

$$\begin{aligned} y(t) &= ce^{At}x_0 + (h(t)*u(t)) \\ h(t) &= Ce^{At}B + D\delta(t) \end{aligned} \tag{4-1}$$

The response $Y(s)$ of the system (with zero initial conditions) to a unit impulse input $\delta(t)$ is $H(s)$, the transfer function representation from the *Transfer Function System Models* section of Chapter 2, *Linear System Representation*. You accordingly term $h(t)$, the inverse Laplace transform of $H(s)$, the impulse response.

The time-response of discrete systems is found directly as a summation of the information from preceding time points in the state and input histories. Using * to indicate discrete convolution, you can express the time domain output as a function of the discrete impulse response:

$$\begin{aligned} y_k &= CA^k x_0 + (h_k * u_k) \\ h_k &= \begin{cases} CA^{k-1}B & (k > 0) \\ D & (k = 0) \end{cases} \end{aligned} \tag{4-2}$$

System Stability: Poles and Zeros

After you have general expressions for the response of a system over time, according to Equations 4-1 and 4-2, you can assess the *stability* of the system. For the purposes of system analysis within Xmath, you define a stable system as one where output does not grow without bound for any bounded input or initial condition. A necessary and sufficient condition for this type of bounded-input bounded-output (BIBO) stability is:

$$\int_0^{\infty} |h(t)| < M < \infty$$

Continuous systems are BIBO stable if and only if all poles of the system are in the left half of the complex plane; discrete systems are BIBO stable if and only if all poles are within the unit circle in the complex plane.

For a *coprime* transfer function $H(q)$ (one having no root cancellations between the numerator and the denominator), the poles are the roots of the denominator of $H(q)$. $H(q)$ is infinite at these values. Values of q for which the numerator of $H(q)$ is zero are termed the *zeros* of the system.

The poles of a system in transfer-function form are identical to the eigenvalues of the A matrix in that system's equivalent state-space representation.

For systems in transfer-function form, zeros are easily defined as the polynomial roots of the numerator. You define the system matrix for a state-space system as

$$S(\lambda) = \begin{bmatrix} \lambda I - A & B \\ -C & D \end{bmatrix}$$

and define the *zeros* of S(λ) as any values of λ for which the system matrix drops rank. For single-input single-output systems this is equivalent to the polynomial zeros of the transfer-function numerator. This definition is somewhat more complex for MIMO systems.

In terms of the dynamic response associated with the poles and zeros of a system, a pole is said to be stable if the response it contributes decays over time. If the response becomes larger over time, the pole is said to be unstable. If the response remains unchanged over time, you describe the pole that causes it as neutrally stable. All the closed-loop poles of a system must be stable to describe the system as stable.

poles( )

```
p = poles(Sys)
```

The `poles( )` function returns a vector listing all the poles of a system. If the input system `Sys` is in transfer-function form, `poles( )` obtains the poles from the roots of the transfer function's denominator (which are automatically stored if the system is in zero-pole format or if the roots have been previously calculated). If `Sys` is in state-space form, the poles are computed as the eigenvalues of the A matrix. To see how to use `poles( )` with a system in transfer function form, refer to Example 4-1.

Example 4-1 Using poles( ) with a System in Transfer Function Form

```
H = 0.5*polynomial([-0.36])/...
    makepoly([1,2.79,2.74,1.11,0.16]);

poles(H)

ans (a column vector) =

-0.395 + 0.0630476 j
-0.395 - 0.0630476 j
-1
-1
```

zeros( )

```
[z,k] = zeros(Sys)
```

The `zeros( )` function finds the *invariant zeros,* the values of λ at which $R(\lambda) = 0$ and $S(\lambda)$ lose rank, and gain is returned only for SISO systems (of a system `Sys`). If `Sys` is in transfer function form, the zeros are obtained

directly from the roots of the transfer function numerator. If `Sys` is in state-space form, the definition of its zeros arises from the system matrix,

$$S(\lambda) = \begin{bmatrix} \lambda I - A & B \\ -C & D \end{bmatrix} \tag{4-3}$$

and its MIMO transfer function:

$$R(\lambda) = C(\lambda I - A)^{-1}B + D \tag{4-4}$$

Defining n as the number of states in the system, p as the number of outputs, and m as the number of inputs, the *normal rank* of S(λ) is n + min(m,p). If the rank of S(λ) equals the normal rank, the system is *nondegenerate*. The values of λ, where $R(\lambda) = 0$ and $S(\lambda)$ loses rank, are the invariant zeros of the system. For degenerate cases in which the normal rank of S(λ) is less than $n + r$, the zeros are defined analogously.

If a system is minimal (that is, no other system with lower order and the same R(λ) exists), these invariant zeros are termed *transmission zeros*. When the matrix in Equation 4-4 loses rank for some value $\lambda = \lambda_0$, there exists a vector $[x_0' \; u_0']'$ of initial states and inputs such that:

$$\begin{bmatrix} \lambda_0 I - A & B \\ -C & D \end{bmatrix} \begin{bmatrix} x_0 \\ u_0 \end{bmatrix} = 0$$

Thus, there exists an initial state x_0 such that the output y is zero for all values of the input function defined over time t as $u_0 e^{\lambda t}$. Such zeros (λ_0) derive the name *transmission zero*, because their effect is to block transmission of the system input to the output.

Note zeros = system zeros = {invariant zeros} ∩ {transmission zeros}.

For an example using `zeros( )` with a state-space system, refer to Example 4-2. For more details on this topic, refer to [Kai80] and [DeS74].

Example 4-2 Using zeros( ) with a State-Space System

```
Sys=system([-2.3,0.01,5.1;0,-0.35,-2;0,2,-.35],
    [1,.25,.25]',[1.34,0,0],0);

[z,k] = zeros(Sys)
```

```
ans (a column vector) =

   -0.98875 + 2.4773 j
   -0.98875 - 2.4773 j

k (a scalar) = 1.34P
```

Algorithm

The algorithm used for state-space zero computation creates a reduced-order $S(\lambda)$, using Householder reflections to do the necessary orthogonal row and column compressions of the state-space matrices. The eigenvalues of this reduced matrix are then found using *QZ*. This method handles the degenerate case and systems with zeros at infinity [EmV82].

Note `zeros( )` also can be used as a matrix building function when used with scalar or matrix input. For more details on this usage, refer to the *MATRIXx Help*.

Partial Fraction Expansion

By inverse Laplace or z-transforming a transfer function, you can identify the impulse response based on knowledge of the system pole and zero locations. The most convenient form to use in doing this is the partial fraction expansion of the transfer function. Each term of the partial fraction expansion has a constant numerator—the *residue*—and a pole term denominator, as shown in the following equation, where p_2 is a repeated pole, and p_4 and p_5 are a conjugate pair:

$$H(q) = C + \frac{r_1}{q+p_1} + \frac{r_2}{(q+p_2)^2} + \frac{r_3}{q+p_3} + \frac{\alpha_4 q + \alpha_5}{(q+p_4)(q+p_5)} + \ldots + \frac{r_n}{q+p_n}$$

Each p_k represents a pole of the system, and the corresponding r_k is the residue at that pole. If p_k is a repeated pole, it has M residues, where M is the multiplicity of the pole. Complex pole pairs have complex residue pairs. If the transfer function contains a constant (or feedthrough) term, this term is represented by the scalar value C in the preceding equation. The values of the residues give the magnitude of the response from the inverse transform of the respective partial fraction terms. For an example of dynamic response with partial fraction expansion, refer to Example 4-3. [Oga70] provides a good reference on partial-fraction expansion for different orders of complex and real poles. [ChB84] contains a thorough mathematical treatment of residues.

Example 4-3 Dynamic Response through Partial Fraction Expansion

To illustrate how you can examine the stability and dynamic response of a system using Xmath, start with the open-loop transfer function system

$$G(s) = \frac{(s+0.5)}{s^2(s+2)(s+10)}$$

You close a unity feedback loop around this system, as shown in Figure 4-1.

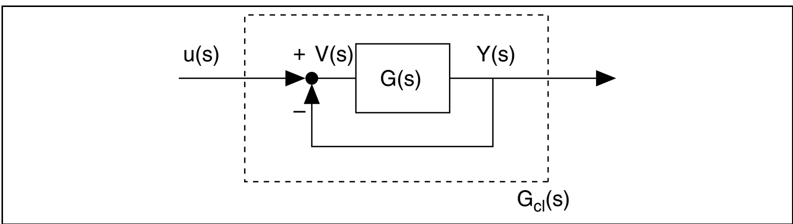


Figure 4-1. Constructing the Closed-Loop System $G_{cl}(s)$ from the Open-Loop System $G(s)$, with Input $U(s)$ and Output $Y(s)$

You can derive the expression for the closed-loop transfer function $G_{cl}(s)$:

$$V(s) = U(s) - Y(s)$$

$$Y(s) = G(s)V(s) \qquad \left(G_{cl}(s) = \frac{Y(s)}{U(s)} = \frac{G(s)}{1+G(s)}\right)$$

Calculate the closed-loop transfer function.

Note You convert the state-space system returned by `feedback( )` to a transfer function using `check( )`.

```
sys = polynomial(-0.5)/polynomial([0,0,-2,-10]);
syscl=feedback(sys);
[,syscl] = check(syscl,{tf, convert})
syscl (a transfer function) =

   (s + 0.5)
---------------------------------------------
                                        2
(s + 1.95266)(s + 10.0118)(s + 0.0354992s + 0.02...

initial integrator outputs
0
0
```

```
0
0
Input Names
-----------
Input 1
Output Names
------------
Output 1
System is continuous
```

You can examine the stability of $G_{cl}(s)$ by representing it as a sum of partial fractions, using the `residue( )` function.

```
residue(syscl)
ans (a pdm) =
                  Poles |
-------------------------+----------------------
 -0.0177496 - 0.158936 j | Order 1  0.0180045 + ...
 -0.0177496 + 0.158936 j | Order 1  0.0180045 -...
 -1.95266                | Order 1 -0.0478224
-10.0118                 | Order 1  0.0118134
```

`residue( )` returns a PDM with the poles as the domain elements, and the associated dependent matrices being the residue at each pole. It also can be expressed in the following form:

$$G_{cl}(s) = \frac{-0.0478}{(s+1.95)} + \frac{0.0118}{(s+10.012)} + \frac{0.036(s+0.01775)+0.16065(0.1589)}{(s+0.01775)^2+(0.1589)^2}$$

Using a table of inverse Laplace transforms to convert this expression to the transient time response rather than a complex frequency response, you can rewrite the time response $G(t)$ as:

$$G(t) = -0.0478e^{-1.95t} + 0.0118e^{-10.012t} + e^{-0.018t}(0.036\ \cos(0.1589t) + 0.160\ \sin(0.1589t))$$

Notice from this example that because all the poles are in the left half plane, the response each contributes is an exponential which decays with time, so this closed-loop system is stable.

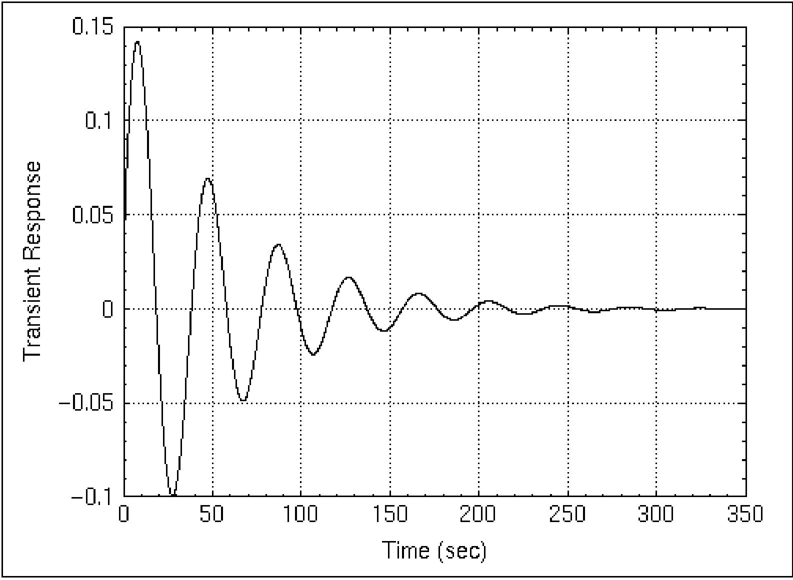


Figure 4-2. Transient Response of the Closed-Loop System as a Function of Time

You also can calculate the impulse response directly with

```
t = [0 : 0.1 : 350 ];
hi = impulse(syscl,t);
plot(hi, { xlab = "Time (sec)", ylab = "Transient 
Response"})
```

Calculating the impulse response gives you the transient response shown in Figure 4-2.

Notice that this response actually takes quite a while to die out because of the small time constants, which correspond to small pole values, in the exponential terms. This is why poles with a small magnitude are frequently called "slow" poles, whereas poles with a large magnitude contribute a response which decays quickly and thus are called "fast" poles.

residue( )

```
[rp,C] = residue(sys,pls,ordr,{isVector,tol})
```

The `residue( )` function calculates the nth-order residue of a transfer-function form system at any of its poles, including Infinity. It returns a PDM `rp` where domain contains the pole locations and where dependent matrices contain the residues corresponding to each pole. `pls` and `ordr` are optional inputs allowing you to specify the pole values

and orders for which the residue(s) should be found. If a user-specified value for `pls` is not actually a pole of the system or if the order requested is greater than the multiplicity of the pole, the corresponding residue is returned as zero. `C` contains the value of the constant term.

Example 4-4 uses the transfer function from Example 2-10, *Verifying a Discretization Using makecontinuous( )*.

Example 4-4 Calculating the Residues of a System

```
G= system(0.5*polynomial([-0.36]),
   polynomial([-1,-1,-0.395+0.06305*jay,
   -0.395-0.06305*jay]));

Rp=residue(G)

Rp (a pdm) =

             Poles |
-------------------+----------------------------
-0.395 - 0.06305 j | Order 1  0.738493 - 0.2277 j
                   | 2      0
-------------------+----------------------------
-0.395 + 0.06305 j | Order 1  0.738493 + 0.2277 j
                   | 2      0
-------------------+----------------------------
-1                 | Order 1 -1.47699
                   | 2       -0.864864
-------------------+----------------------------
```

combinepf( )

```
Sys = combinepf(Rp,C,{var})
```

`combinepf( )` reverses the operation performed by `residue( )`, combining partial fractions into a single transfer function. It expects a PDM of the form shown in Example 4-4 as input.

Use `combinepf( )` to convert partial fractions to a transfer function.

Using the variable `Rp` you obtained in Example 4-4:

```
G2=combinepf(Rp, {var = "s"})

G2 (a transfer function) =

       0.5s + 0.18
---------------------------
      2  2
```

```
(s + 1) (s + 0.79s + 0.16)

initial integrator outputs
0
0
0
0
Input Names
-----------
Input 1

Output Names
------------
Output 1

System is continuous
```

Note `G2` matches the system `G` where residues were computed in Example 4-4.

General Time-Domain Simulation

When modeling a dynamic system and trying to determine its response to the input values it is likely to receive in use, you generally will want to simulate system behavior with more general input signals than the zero or step inputs used in `initial( )`, `impulse( )`, and `step( )`. To do this, use the `*` operator between a dynamic system object and a PDM containing the input data you want to use in the simulation.

Borrowing from the standard frequency response notation for a system where:

$$y(s) = H(s) \times u(s)$$

Xmath defines the operation system*PDM as a time domain simulation. Thus for any dynamic system `Sys` (continuous or discrete) and for a PDM u representing the external stimulus as a function of time, the operation y = `Sys` $\times$ u creates a PDM y which contains the outputs of the system as a function of time.

For a dynamic system with n_y outputs and n_u inputs, the input vector is defined to be $n_y \times 1$ and the output vector is $n_y \times 1$. Thus the input PDM u should be $m \times 1 \times p$, where p is the number of time points in u.

Often it is desirable to run several simulations with different inputs. In this case, you can define a PDM whose columns contain the input vectors for the different simulations. Then u will be $n_y \times q \times N_{samp}$, where q is the number of different simulations to be run. The resulting y will be $n_y \times q \times N_{samp}$, with each column of the PDM corresponding to a different simulation.

The input PDM must have a regular domain—that is, the interval between each domain value and the one succeeding it must be the same over all points in the domain. If the system is discrete, the domain intervals must be equal to the system's sampling period. If the system is continuous, it is discretized using the exponential (zero-order hold) method, with the sampling interval set equal to the input domain interval spacing.

Note For accurate results, you need to make sure this sampling interval is small enough that discretization effects are negligible.

The next step is to create a general signal and store it as a PDM where domain is time as shown in Example 4-5. Because you are using a SISO system, this input is a single-channel PDM.

Example 4-5 Performing a General Time-Domain Simulation

```
t = 0:0.1:15;
osig = ones(1,30);
sig = [0*osig,0.5*osig,osig,0.5*osig,0*ones(1,31)];
U = pdm(sig,t);
```

Create the system:

```
Sys = system([-2.3,0.01,5.1;0,-0.35,-2;0,2,-.35],
    [1,.25,.25]',[1.34,0,0],0);
```

and perform the simulation:

```
Y = Sys*U;
```

To see how well the system tracks the input signal, plot the input, as follows, and the system's response, shown in Figure 4-3.

```
plot ([U,Y], {legend = ["Input Signal",
    "System Response"],line_color = "black",
    xlab = "Time (sec)", ylab = "Amplitude"})
```

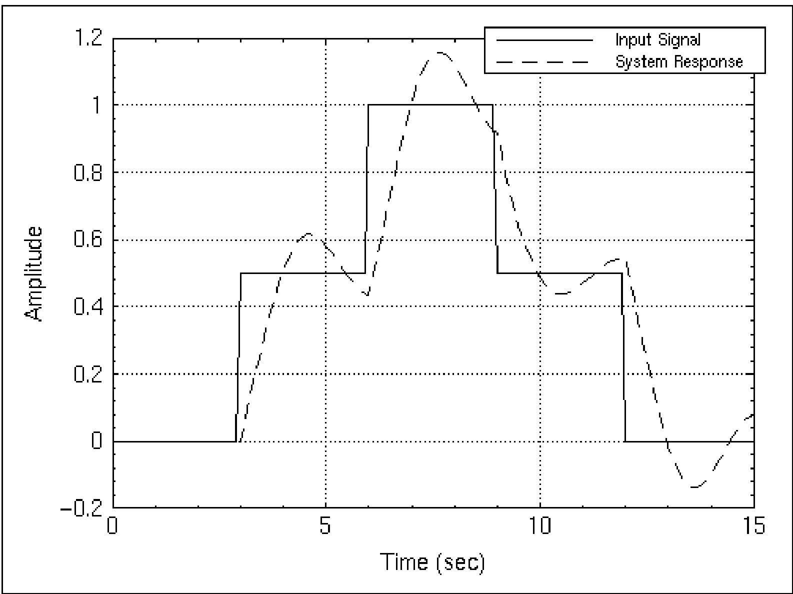


Figure 4-3. System Time Response to a Series of Step Signals

The (system)*(PDM) construct for performing time-domain simulation is used analogously no matter how many inputs the system has. For a multi-input, multi-output system, the number of rows of the input PDM must match the number of inputs of the system. For an example of general time-domain simulation for a MIMO system, refer to Example 4-6.

Example 4-6 General Time-Domain Simulation for a MIMO System

```
sys = system([0,1,0;0,0,1;-2,-4,-3],
    [0,1;1,0;-1,1],[0,1,-1;1,2,1],[]);

u = pdm([sin(-3*pi:0.01:3*pi);
    cos(-3*pi:0.01:3*pi)],{rows=2,columns=1});

y = sys*u;
```

Impulse Response of a System

An impulse input to a system is defined somewhat differently depending on whether the system is discrete or continuous. For a continuous-time system, an impulse is a unit-area signal of infinite amplitude and infinitely small duration occurring at time $t = 0$, and having value zero at all other times. For a discrete system, an impulse can be thought of as a physical pulse which has unit amplitude at the first sample period and zero amplitude for all other times.

The Laplace transform of the continuous-time impulse—often referred to as $\delta(t)$—is 1. Thus, the Laplace transform of a output of a system to a unit impulse is merely its transfer function $H(s)$, as discussed in the *Time-Domain Solution of System Equations* section.

A similar definition, using the z-transform, can be made for the discrete-time impulse response. However, the values of the impulse response of a discrete system also have the property that they define the Markov parameters for that system. Based on the state-space representation of the system, these parameters are defined as the values

$$h_i = \{CA^{i-1}B, i = 1,2,\ldots\}$$

These parameters are uniquely determined by the transfer function of the system [Kai80]:

$$H(z) = C(zI - A)^{-1}B = \sum_{i=1}^{\infty} h_i z^{-i}$$

and they also are the terms of the discrete impulse response.

impulse( )

```
y = impulse(Sys,t)
```

The `impulse( )` function computes the impulse response of a dynamic system. The time vector, t, is an optional input. If not specified, a default time range will be computed using `deftimerange( )`. Refer to the *deftimerange( )* section. For a continuous-time system, the impulse response is calculated at each point in the time vector. For a discrete system, the first n Markov parameters are returned, where n is the length of the time vector (which must be regularly spaced).

Note A continuous system and its discrete-time equivalent (computed using the impulse-invariant z-transform) have impulse responses differing only by a factor of 1/*dt*.

`impulse( )` computes the impulse response by using the B matrix from the system's state-space representation as the initial conditions. A system with n_i inputs has n_i initial conditions, each of which is set up as a column of the B matrix. The impulse response is then a time-domain simulation of the system using an appropriately-sized zero input.

The output *`y`* is a PDM where domain is the time vector `t`. Each dependent matrix in *`y`* has as many rows as there are outputs of `Sys`, and as many columns as there are inputs of `Sys`. Thus the (*`i,j,k`*) element of *`y`* is the impulse response at output *`i`* from input *`j`* at time *k*. In Figure 4-4, where all the poles of this continuous system are stable (in the left half of the complex plane), the impulse response eventually dies out to zero. For an example of a 15-second impulse response of a stable state-space system, refer to Example 4-7.

Example 4-7 15-second Impulse Response of a Stable State-Space System

```
Sys = system([-2.3,0.01,5.1;0,-0.35,-2;0,2,-.35],
    [1,.25,.25]',[1.34,0,0],0);
Yt = impulse(Sys,0:0.1:15);
plot (Yt, {xlab = "Time (sec)",
    ylab = "Amplitude"})
```

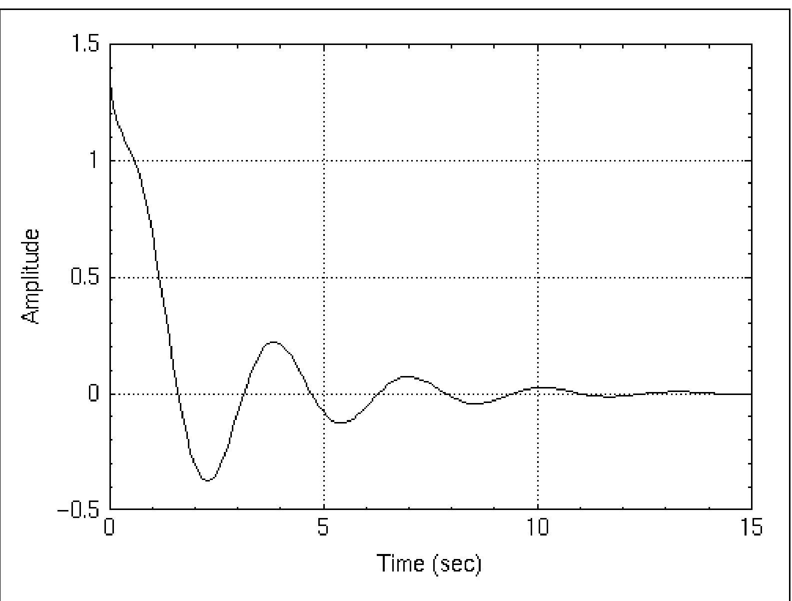


Figure 4-4. 15-Second Impulse Response

deftimerange( )

```
tvec = deftimerange(Sys)
```

`deftimerange( )` computes a regular time vector (in units of seconds) that can be used in time-domain simulations to observe the effects of all or most of the input system's dynamics, as indicated by pole and zero location.

Within `deftimerange( )`, the poles of the system are obtained using `poles( )`. For continuous-time systems, the poles are scaled by a factor of $1/2\pi$ (to convert from radians) and the time constant (in seconds) is obtained as the reciprocal of four times the value of the pole with the maximum absolute value (the "fastest" pole). For discrete-time systems, the logarithm of the poles is taken and scaled by the sampling interval. The sampling interval is automatically used as the step size for the `tvec` time vector. If all system poles are integrators, the step size defaults to 0.01.

The maximum time, `Tmax`, is computed as follows, with `vP` denoting the vector of scaled poles and `dt` the period:

```
Tmax=abs(log(.05)/...
    max(real(vP(find(real(vP)<>0)))))

If Tmax == null # all poles purely imaginary

    Tmax=100*dt

endIf

Tmax=max(Tmax,10*dt)

tvec=0:dt:Tmax
```

Though `deftimerange( )` calls `minimal( )` to remove any pole-zero cancellations, it does not consider the location of the system zeros in computing the time vector. As a result, if `Sys` has zeros that are more than a decade beyond its maximum or minimum poles, the effects of these zeros may not be apparent in a time response calculated using `tvec`. You should supply your own time vector to `impulse( )`, `initial( )`, and `step( )` in these cases.

System Response to Initial Conditions

It is often assumed that the states of a system have zero initial conditions, and the `X0` field of a state-space system object correspondingly defaults to zero. In many cases, however, you need to examine the system response to a given set of nonzero initial conditions; a common system design goal is that this response become zero (or negligibly small) as quickly as possible. The Xmath function `initial( )` allows you to do this. You also can use superposition to calculate forced initial condition response.

initial( )

```
Y = initial(Sys,T,{X0})
```

The `initial( )` function computes the unforced response of a system from its initial conditions. By default it uses the initial conditions stored with the input system itself, but an alternate set of initial conditions can be specified as the keyword `X0`. A time vector (with spacing equal to the sampling period of the system if it is discrete) also can be specified, or `initial( )` can compute a default time vector internally using `deftimerange( )`.

The simulation performed in `initial( )` uses an input of zero for each point in the time vector. The output `Y` is a PDM where domain is the time vector.

By varying the initial values of the states individually, you can determine which is the most sensitive. For an example using `initial( )` to determine the sensitivity of the states, refer to Example 4-8.

Example 4-8 Using initial( ) to Determine the Sensitivity of the States

```
Sys = system([-2.3,0.01,5.1;0,-0.35,-2;0,2,-.35],
    [1,.25,.25]',[1.34,0,0],0);
ic1 = initial(Sys, 0:.1:15, {X0 = [1,0,0]});
ic2 = initial(Sys, 0:.1:15, {X0 = [0,1,0]});
ic3 = initial(Sys, 0:.1:15, {X0 = [0,0,1]});
plot ([ic1,ic2,ic3], {xlab = "Time (sec)",
    legend = ["State 1", "State 2", "State 3"],
    ylab = "Amplitude"})
```

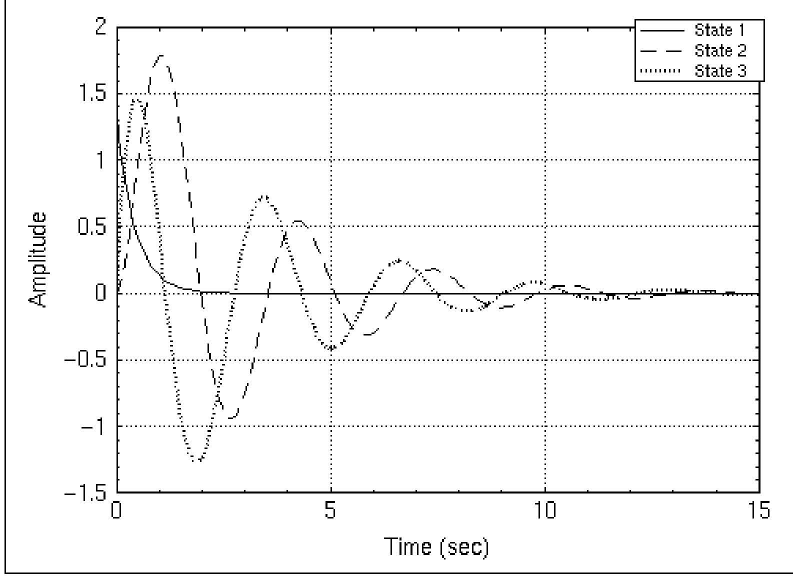


Figure 4-5. 15-Second System Response to Unity Nonzero Conditions at Each of the States

In Figure 4-5, notice that the value of the second state has the highest maximum value and takes the longest to “zero out.”

Step Response

The response of a system to a unit step input is one of the most commonly used measures of how well a given control system's output tracks the system input. A unit step is a time signal which is zero up until the beginning of the time period of interest, and one thereafter. This indicator is popular because it is easy to compute and interpret. It also is mathematically possible to calculate the response to any input if the response to a unit step is known. The performance measures associated with the step response are as follows:

- Delay time (t_d)—The time required for the response to reach half its final value.
- Rise time (t_r)—The time required for the response to rise from 10% of its final value to 90% of its final value.
- Peak time (t_p)—The time required for the response to reach the peak value of its first overshoot.
- Maximum overshoot (M_p)—The response value which most exceeds unity, expressed as a percent.
- Settling time (t_s)—The time required for the response to reach 5% of its final value.

These performance measures are obtained easily with a few lines of MathScript, as demonstrated in Example 4-9. For a plot of these performance measures, refer to Figure 4-6.

step( )

```
Y = step(Sys,T)
```

The `step( )` function computes the unit step response of a dynamic system over a time period which can be specified with the optional time vector `T`. If `T` is not specified, `step( )` computes a default time vector using `deftimerange( )`.

The output, `Y`, is a PDM where domain is the time vector and dependent matrices have row size equal to the number of inputs and column size equal to the number of outputs.

Example 4-9 Performance Measurements for a Step Response

```
Y = step(Sys, 0:.1:15);

plot (Y,{x_lab = "Time (sec)",
    ylab = "Amplitude", xinc=1, yinc=.1})
```

From Figure 4-6 you see that the delay time (t_d) is about 0.5 seconds, the rise time (t_r) is 0.8 seconds, the peak time (t_p) is 1.6 seconds, the settling time (t_s) is about 5.5 seconds, and the maximum overshoot (M_p) is about 24%.

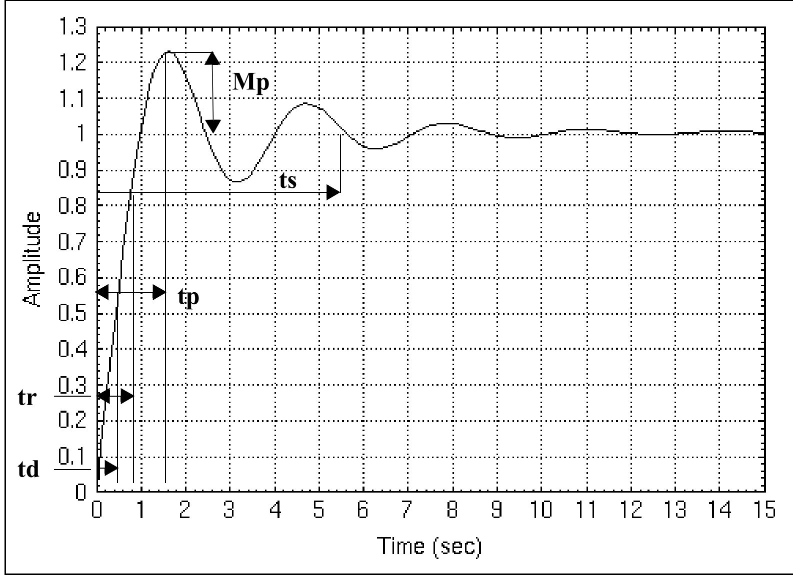


Figure 4-6. 15-Second Step Response, Showing Performance Measures

You can compute these values from the 151-point step response data vector `Y` and substantiate your estimates.

First, you find the final value of the response:

```
Yf = makematrix(Y(151));
```

Get indices of all values > half the final value:

```
gt_half = find(Y > 0.5*Yf);
```

Time corresponding to first index in `gt_half`:

```
td = domain(Y(gt_half(1,1)))
```

```
td (a scalar) = 0.5
```

Get indices of all values > 0.1 * final value:

```
gt_1_10 = find(Y > 0.1*Yf);
```

Get indices of all values > 0.9 * final value:

```
gt_9_10 = find(Y > 0.9*Yf);
```

Subtract domain values to get time duration:

```
tr = domain(Y(gt_9_10(1,1)))-...
    domain(Y(gt_1_10(1,1)))
```

```
tr (a scalar) = 0.8
```

Get peak value of response:

```
maxY = max(Y, {channels});
```

Index and time corresponding to peak value:

```
maxtp = find(Y == maxY);
tp = domain(Y(maxtp(1)))
```

```
tp (a scalar) = 1.6
```

Convert peak value to a percentage > 1:

```
Mp = round(10000*(maxY-1))/100
```

```
Mp (a scalar) = 23.21
```

Reformat the step response in reverse time:

```
Yrev = Y(151:-1:1);
```

Response values within 0.05 * final value:

```
gt_05 = find(Yrev <= 0.95*Yf | Yrev >= 1.05*Yf);
```

Time value of first point within bound:

```
ts = domain(Yrev(gt_05(1,1)))
```

```
ts (a scalar) = 5.1
```

5

Classical Feedback Analysis

The open-loop systems analyzed in Chapter 4, *System Analysis*, generally perform in a satisfactory manner only if the system model is very accurate and there are no external disturbances. These conditions usually are not met. Feedback presents an effective way to control the output of a system. The functions in this chapter address the problem of suitably controlling an open-loop plant through output feedback. They are most often applied to single-input, single-output (SISO) systems. With the exception of `rlocus( )` and `bode( )`, these functions also can be used with multi-input, multi-output (MIMO) systems.

Feedback Control of a Plant Model

The key principle of feedback is that the output of a system be fed back, compared to a reference or "desired" output value, and then the error between the two terms used to correct the system's output so that it matches the reference. The basic diagram of a feedback control system is shown in Figure 5-1.

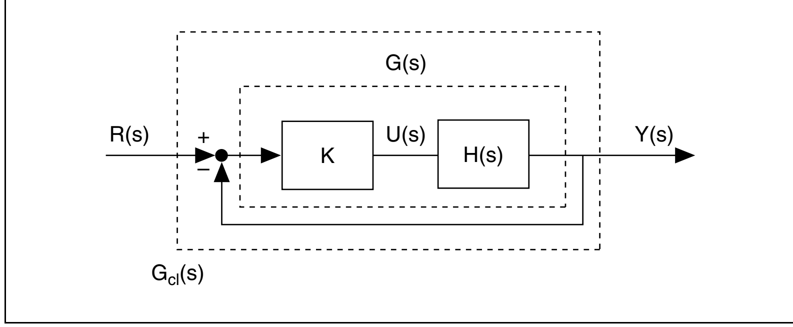


Figure 5-1. Feedback Control System Block Diagram

The output of the open-loop system is $KH(s)$; the output of the closed-loop system shown in Figure 5-1 is given by:

$$\frac{Y(s)}{R(s)} = \frac{KH(s)}{1 + KH(s)}$$

Because open-loop systems are generally easier to study and model than closed-loop systems, you want to design closed-loop systems based on information obtainable from the open-loop system.

Root Locus

In Chapter 4, *System Analysis*, you learned how the location of the system poles and zeros affects the stability of the system, so an effective feedback control design should take into account the closed-loop pole and zero locations. If you represent the open-loop transfer function $H(s)$ as the quotient of the numerator and denominator as follows:

$$H(s) = N(s)/D(s)$$

you can rewrite the characteristic equation of the closed-loop system as follows:

$$1 + KH(s) = D(s) + KN(s) = 0$$

This restates the fact that the open-loop system poles (which correspond to $K = 0$) are the roots of the transfer function denominator, $D(s)$. As K becomes large, the roots of the previous characteristic equation approach the roots of $N(s)$—the zeros of the open-loop system—or infinity. For a closed-loop system with a nonzero, finite gain K, the solutions to the preceding equation are given by the values of s where both of the following are true:

$$|KH(s)| = 1 \qquad \angle\ H(s) = \pm(2k+1)\pi \qquad (k = 0, 1, \ldots)$$

The *root locus* is a plot in the real-imaginary axis showing the values of s that correspond to pole locations for all gains K, starting at $K = 0$ (the open-loop poles) and ending at $K = \infty$.

Root locus plots provide an important indication of what gain ranges can be used while keeping the closed-loop system stable. As discussed in the *System Stability: Poles and Zeros* section of Chapter 4, *System Analysis*, continuous-time systems are stable as long as their poles are in the left half of the s-plane (have a negative real part) and discrete-time systems are stable as long as their poles remain within the z-plane unit circle.

The Xmath root locus-plotting utility exists for SISO systems only, though either state-space or transfer function models can be specified.

rlocus( )

```
rlroots=rlocus(sys,K,{xmin,xmax,ymin,ymax,pattern,graph})
rlocus(sys,{xmin,xmax,ymin,ymax,pattern})# (interactive)
```

The `rlocus( )` function computes and draws root locus diagrams for continuous-time and discrete-time SISO systems. The first syntax, in which a vector of gain values is specified, generates a plot showing the closed-loop pole locations for each gain. In the **Graphics** window, the complete locus is drawn as a solid line, with `o`s marking the location of zeros and `x`s delineating open-loop pole locations. The second syntax brings up a window through which you can interactively modify the closed-loop gain and see the corresponding pole locations change on the locus.

A grid showing pole stability range can be invoked with the `pattern` keyword. The optional keywords specifying maximum and minimum *x* and *y* values can be used to restrict the range of the selected s- or z-plane. These can be changed interactively if the interactive syntax is used. Click **RECOMPUTE** to activate rate changes.

Example 5-1 shows how to plot the rool locus created in Example 2-9, *A Comparison of Several Discretization Methods*.

Example 5-1 Plotting a Root Locus

```
H = system(0.5*polynomial([-0.36]),
    makepoly([1,2.79,2.74,1.11,0.16]));
```

You can create and graph a root locus, scaling the range of the real-imaginary plane as follows:

```
rlocus(H, {xmin=-2, xmax=0, ymax=0.5, ymin=-0.5})
```

These functions give the results shown in Figure 5-3. The large `x`s on the plot correspond to the open-loop pole locations you found for this system in Example 4-1, *Using poles( ) with a System in Transfer Function Form*, and the zeros correspond to the single zero at –0.36.

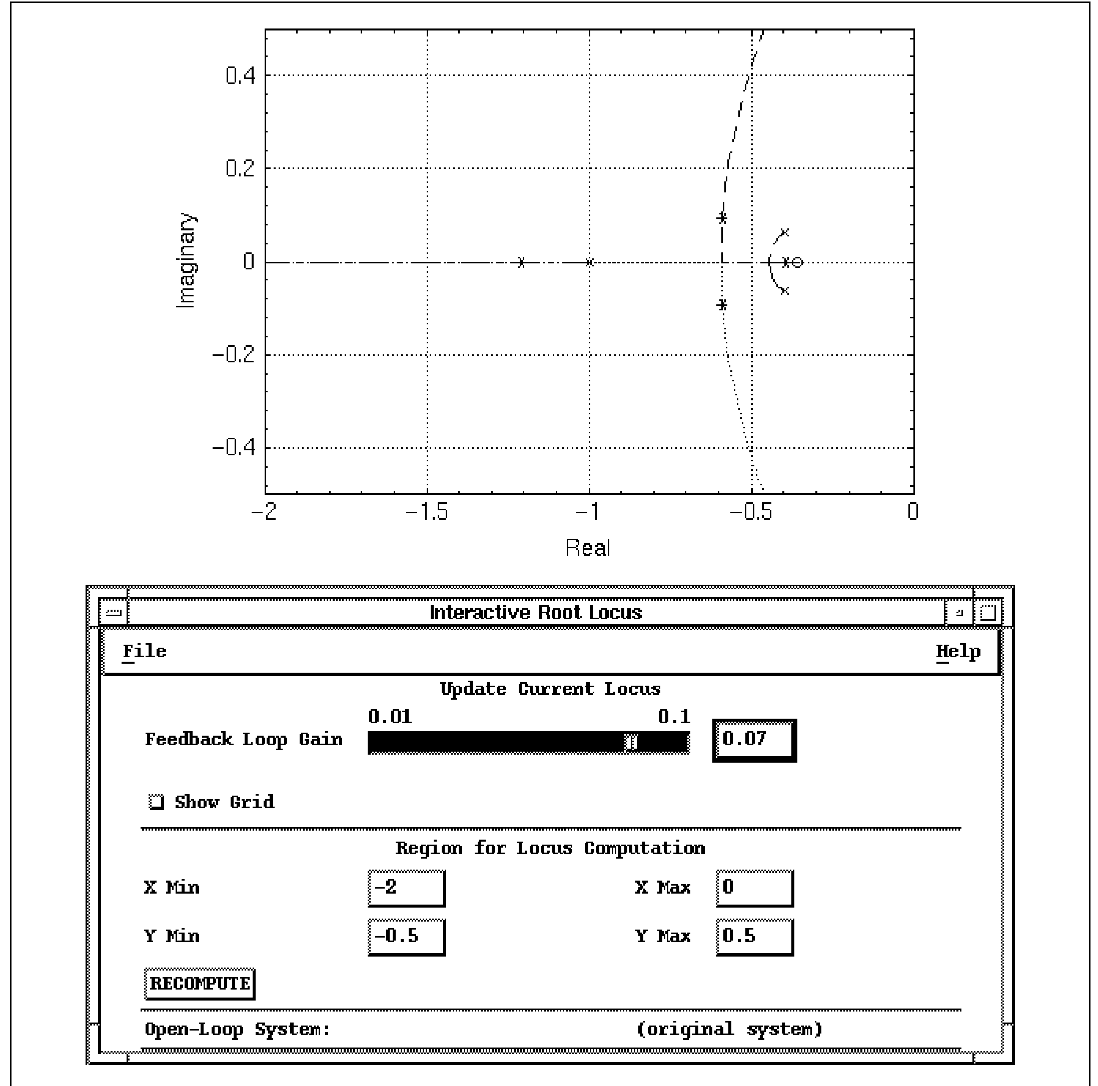


Figure 5-2. Root Locus of *H* for Gain *K* = 0.07

This syntax allows you to vary the root locus gain through an interactive form. Within this form, you can change the gain value through either a slider or an editable label where value corresponds to the current slider position. The slider range is automatically updated when the slider is moved to its maximum or minimum value, or when a gain value outside the current slider limits is entered into the editable label.

As the gain varies, small $*$'s appear on the locus indicating the closed-loop pole location for that choice of gain. The locus shown in Figure 5-2 shows that for small gain values the closed-loop system is stable, with all of its roots in the left half of the complex plane.

Frequency Response and Dynamic Response

The frequency response of a dynamic system is the output, or response, of a system given unit-amplitude, zero-phase sinusoidal input. A sinusoidal input with unit amplitude and zero phase, and frequency ω produces the following sinusoidal output:

$$H(j\omega) = A(\omega)e^{j\phi(\omega)}$$

where A is the magnitude of the response as a function of ω, and ϕ is the phase. The magnitude and phase of the system output will vary depending on the values of the system poles, zeros, and gain. In many practical engineering applications, the system poles and zeros are not precisely known. Because the frequency response can be determined experimentally, undesirable parts of the system's frequency response then can be improved by adding known compensation to the system.

freq( )

```
H=freq(Sys,F,{Fmin,Fmax,npts,track,delta})
```

The `freq( )` function calculates the frequency response of a system in several different ways, depending on the system representation. For continuous-time transfer functions, the frequency response $H(\omega)$ at a given frequency ω is obtained by substituting the complex frequency value $j\omega$ for $q$in the following equation. For discrete-time transfer functions, the value e^{jwT}, with T the system sampling interval, is substituted for q instead.

$$Sys(q) = \frac{(q+z_1)\ldots(q+z_m)}{(q+p_1)\ldots(q+p_n)}$$

For continuous-time state-space systems, the basic method for finding frequency response is to substitute different frequency values, represented by ω, into the following equation:

$$H(jw) = C(jwI - A)^{-1}B + D$$

For discrete-time state-space systems with a sampling interval of *T*, the frequency response for each frequency point ω is shown in the following equation:

$$H(jw) = C(e^{jwT}I - A)^{-1}B + D$$

Algorithm

The algorithm, based on [Lau86], uses a Hessenberg decomposition to simplify the previous equations and is quite robust. It finds matrices *P* and *H* such that *A* = *PHP*', where PP' = *P*'*P* = *I* and *H* is a Hessenberg matrix, and substitutes for *A*. Because *H* is zero only below the first subdiagonal, the number of operations needed to evaluate the response expression is proportional to the square of the size of *A*.

`freq( )` allows you to prespecify frequency ranges of interest, or it can generate a representative frequency range from minimum and maximum frequencies you specify. It then evaluates the complex frequency response over those frequencies, using specialized algorithms to do this efficiently.

You can specify either a complete set of frequency points (the optional input `F`) or a range of frequency points (the keyword pair `Fmin` and `Fmax`) at which to evaluate the response. The `track` keyword indicates that phase tracking will be used to determine the values of the frequencies between `Fmin` and `Fmax`. The number of intermediate frequency points produced using `track` varies depending on the system and the `Fmin` and `Fmax` you choose. Alternately, you can use the `npts` keyword to specify the exact number of logarithmically-spaced frequency points you want computed. Specifying `track` invokes an algorithm which tracks the phase of the frequency response to make sure that all peaks and valleys are included in the computed response. The `delta` keyword indicates the amount of phase change (measured in degrees) to which the response evaluation should be sensitive. If phase change between two adjacent frequency points exceeds this delta, closer frequencies are used until either the phase change is less than `delta` or a maximum number of iterations is reached. Evaluation is forced at key frequency points which include the poles and the points lying halfway between adjacent poles.

`freq( )` returns a PDM having the frequency range as its domain. The dependent matrices of the frequency response PDM have as many rows as the system has outputs, and as many columns as the system has inputs. For MIMO systems, the (*i*,*j*) element of a dependent matrix is thus interpreted as the frequency response from input *j* to output *i*. This frequency response forms the core of the classical control design tools discussed in this chapter.

For an example of frequency response of a simple system, refer to Example 5-2.

Given the single-input, single-output open-loop plant in Figure 5-3, where $U(s)$ and $Y(s)$ are the frequency domain input and output, respectively, you can examine its response characteristics and see how you can improve them using the frequency-response based control design functions in this chapter.

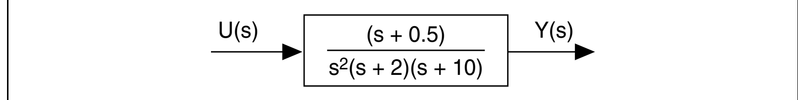


Figure 5-3. Representation of the Open-Loop System

Example 5-2 Frequency Response of a Simple System

You can create the system directly in transfer function form:

```
sys = system(polynomial(-0.5),
   polynomial([0,0,-2,-10]));
```

and then obtain the frequency response directly:

```
H = freq(sys,{Fmin = 0.01,Fmax = 10,npts = 150});
```

`freq( )` also can be called with a predefined vector of frequency points, or you can specify that phase tracking be used to compute frequency points between the minimum and maximum frequencies. The number of frequency points used with tracking will vary. To illustrate:

```
H = freq(sys,{Fmin=0.01,Fmax=10,track,delta=.5});
```

```
size(H)
```

```
ans (a row vector) = 1    1    335
```

The dynamics of this system are adequately reflected in both frequency responses. However, systems having more closely-placed pole and zero locations are good candidates to use with the `track` keyword.

Bode Frequency Analysis

While `freq( )` provides you directly with the frequency response, other tools in the Control Design module can give you more insight into what the open- and closed-loop frequency responses of a system imply about the system behavior. Bode plots of system frequency response are useful

because they can be used to assess the relative stability of a closed-loop system given the frequency response of the open-loop system. It should be noted that the open-loop system should be stable and *minimum phase*, having no right-half plane poles or zeros, for this type of analysis [Oga70].

The following complex frequency response:

$$H(\omega) = A(\omega)e^{j\phi(\omega)}$$

can be separated into two parts, which are both functions of the frequency:

- ω: the magnitude, $A(\omega)$
- the phase, ϕ

The magnitude can be obtained as the absolute value of the response, whereas the phase is obtained from the four-quadrant arctangent of the response.

The standard Bode format comprises two subplots:

- The upper plot shows the decibel gain (the common logarithm of the magnitude, multiplied by 20) plotted against the logarithm of the frequency. Logarithmic (decibel) plots are a particularly useful tool for indicating magnitude response because the multiplication of magnitudes is shown as the sum of their logarithms, thus allowing you to determine the system response with varying gains quickly.
- The lower plot shows the phase, in degrees, as a function of the logarithm of the frequency. For both the gain and phase plots, logarithmic frequency scaling is used because it allows a wide range of frequency-dependent behavior to be displayed simultaneously.

Because the gain and phase plots are additive for systems cascaded in series, Bode plots of an open-loop plant and potential compensators can be added to determine the frequency-response characteristics of the complete system. The plots also illustrate system bandwidth, as the frequency at which the output magnitude is reduced by three decibels, or attenuated to approximately 70.7% (a factor of $(\sqrt{2}/2)$ of its original value.

Bode plots also provide an important aid to evaluate how stable—or, more specifically, how close to instability—a closed-loop system is. As discussed in the *System Stability: Poles and Zeros* section of Chapter 4, *System Analysis*, for the continuous case, the closed-loop poles of a stable system lie in the left half of the complex plane.

Referring to the entire closed-loop system in Figure 5-1 as G_{cl}, the poles of G_{cl} are the roots of its denominator—that is, the values of s such that either of the following is true:

$$1 + G(s) = 0$$
$$G(s) = -1$$

The magnitude (absolute value) of $G(s)$ is 1 at each pole of $G_{cl}(s)$, and the phase (given by the four-quadrant arctangent) of $G(s)$ is –180° at each pole of $G_{cl}(s)$. For any neutrally stable system, the frequency response magnitude will be equal to 1 (or 0 dB) and the phase will be –180° at the frequency at which the closed-loop roots fall on the imaginary axis.

This analysis is often applied to systems where $G(s)$ consists of a gain, K, and a dynamic model, $H(s)$, in series (as shown in Figure 5-1). For cases in which increasing the gain leads to system instability, the system will be stable for a given value of K if the magnitude of $KH(s)$ is less than 1 at any frequency at which the phase of $KH(s)$ is 180° [FPE87]. You can measure how close a system is to instability by examining the value of the magnitude and phase at these critical values. These measures are termed the gain margin and the phase margin. These are important because real-life models are prone to uncertainties and changes in gain or phase. Typically, systems become unstable with gains that are too high or have too much phase lag. Refer to Example 5-4.

The gain margin indicates by how much the gain can be raised before the closed-loop system becomes unstable. This critical gain value at which instability results can be thought of in several ways. As described previously, this gain value results in the closed-loop poles of the system being located on the imaginary axis. In terms of the Nyquist stability criterion, discussed in more detail under `nyquist( )`, this is the gain value at which the Nyquist plot crosses the negative real axis, where the phase is –180 degrees. The gain margin itself is the reciprocal of this value, expressed in decibels.

The Bode plot provides a clear visual interpretation of the gain margin as the number of decibels by which the gain exceeds zero when the phase equals –180 degrees. The phase margin is the difference between the phase at the point where the response crosses the unit circle (has unit magnitude, or a gain of 0 decibels) and –180 degrees. These margins provide a measure of how near the closed-loop system roots are to instability. Depending on the complexity of the system, there may be multiple gain and/or phase margins.

Referring to Figure 5-4, notice the additional lines drawn on the plots at the frequencies where the gain crosses the 0 dB line and where the phase crosses the 180° line. When the gain crosses the 0 dB line, the phase is about –168°, 12° away from –180°. So the phase margin is approximately 12°. Similarly, when the phase crosses the –180° line, the gain is about –44 dB (44 dB from the 0 dB line), and thus the gain margin is 44 dB.

bode( )

```
[H,dB,Phase] = bode(Sys,{F,keywords})
```

The `bode( )` function uses `freq( )` to compute the frequency response of a system. By default, the `freq( )` keyword `track` is on, but it can be overridden. Refer to the *freq( )* section for more details. When the frequency response `H` is found the decibel magnitude and the phase angle in degrees are computed as follows:

```
dB=20*log10(abs(H); phase=(180/pi)*atan(H)
```

`bode( )` then produces the standard Bode format plots showing response magnitude and phase as functions of frequency. Unlike `freq( )`, `bode( )` does not require a frequency range or a pair of maximum and minimum frequencies; if no range is specified, it uses `deffreqrange( )` to calculate a default frequency range.

`bode( )` often generates more than one set of plots. For MIMO systems, a plot is made for each output with multiple curves, one per input. If there are multiple outputs, a menu will appear which allows you to select an input to view.

If you want to see the response of the system from Example 5-2 to input frequencies ranging from 0.01 Hz to 10 Hz, you can analyze a frequency response using `bode( )`, as shown in Example 5-3.

Example 5-3 Analyzing a Frequency Response Using bode( )

```
sys = polynomial(-0.5)/polynomial([0,0,-2,-10]);
```

```
[H,dB,phase]=bode(sys,
   {Fmin = 0.01,Fmax=10,npts = 300,!wrap})
```

You obtain the gain and phase plots as shown in Figure 5-4.

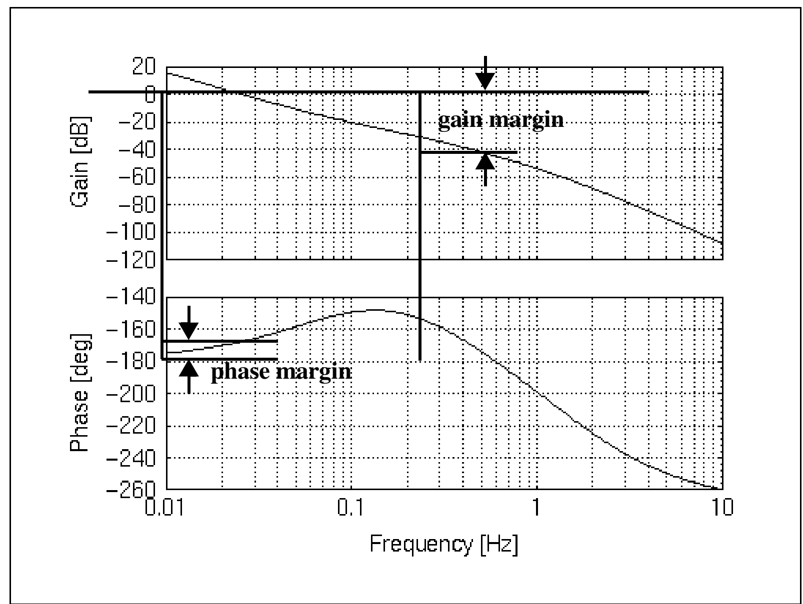


Figure 5-4. Bode Plot Showing System Gain and Phase Margins

These plots illustrate how the location of the system poles and zeros shapes the gain and phase curves. Each pole contributes a factor of –20 dB per decade (frequency interval from ω to 2ω). The two poles at zero cause the magnitude response of the system to start with a slope of –40 dB/decade. The zero at 0.5 radians/sec (about 0.08 Hz) contributes a factor of approximately 20 dB. These gain magnitude factors add, so the slope of the gain plot changes from –40 dB/decade to about –20 dB/decade until you begin to see the influence of the poles at 2 radians/sec. (0.318 Hz) and 10 radians/sec (1.59 Hz), each of which contribute another –20 dB/decade to the slope of the magnitude plot.

The phase is a function only of the pole and zero locations. Notice that in creating the phase plot with `bode( )`, you specified the `!wrap` keyword. This created a phase plot where range goes down to the full angle value of the phase, rather than wrapping the phase between ±180°. Each pole at zero contributes –90° of phase.

The remaining poles are called first-order poles because they are of the following form:

$$s + p_n$$

Each of these contributes a phase angle ϕ defined by:

$$\phi = \operatorname{atan}(\omega / p_n)$$

with ω and p_n expressed in the same units, either radians per second or Hz, and using a four-quadrant arctangent function similar to that provided by `atan2( )` in Xmath. Thus the amount of phase contributed by a first order pole at the frequency

$$\omega = p_n$$

(generally termed the corner frequency, because the asymptotes used to draw different portions of the response intersect and form a corner) is –45°. At frequencies beyond the corner frequency, the phase angle contributed by that pole comes closer and closer to –90°. First-order zeros contribute phase angle in the same manner except that the sign of the angle is positive.

margin( )

```
[gnMargin,phMargin,dPdF,dGdF] = margin(H)
```

The `margin( )` function is a useful tool for evaluating the stability margin of a given system based on its frequency response. It returns PDMs indicating the gain margin and the phase margin, as well as the rate of change of gain and phase.

`margin( )` is defined for SISO systems only. It takes as input either a single PDM representing frequency response or a pair of PDMs containing gain information in decibels and phase information in degrees. In either case, the domain of the input is the set of frequency points, ω.

Within `margin( )`, as within `bode( )`, the frequency response is converted to decibel magnitude and degree phase. All angles are converted to four-quadrant angles between 0° and 360°. Use the following notation for each point i in the frequency range:

$$\Delta x = x(i+1) - x(i)$$

`margin( )` loops over all the frequency points in the response and performs the following computation for phase and gain margins at each, denoting gain margin as *Mg* and phase margin as *Mp*:

$$Mp(i) = phase(i) + \frac{\Delta phase}{\Delta\omega}\left(\Delta\omega - gain(i+1)\frac{\Delta\omega}{\Delta gain}\right)$$

$$Mg(i) = -\frac{(gain(i) + \Delta gain)}{\Delta phase}\left(\Delta\omega - phase(i+1)\frac{\Delta\omega}{\Delta phase}\right)$$

This loop finds all the frequency intervals within the response which contain –180° phase crossings and 0 decibel gain crossings. `margin( )` then interpolates to find more exact frequency values for the crossings. A gain margin value is returned for every pair of phases between which a –180° phase value must occur, and a phase margin is returned for each pair of gains between which a zero-decibel gain value must occur.

`margin( )` also computes the frequency-rate of change for both the phase and the gain of the response.

–180° and 0 dB crossings are difficult to detect accurately if the points in the frequency response are too widely spaced.

You can examine the gain and phase margins of your open-loop system quickly using `margin( )`, without having to draw the bode gain and phase response plots first. Input to margin is the frequency response `H` of the system. Referring to the system defined in Example 5-3, you can see that you already have `H` as the output from `bode( )`, but you can calculate it explicitly using `freq( )` as shown in Example 5-4.

Example 5-4 Obtaining Gain and Phase Margin Using margin( )

```
H = freq(sys,{Fmin=0.01,Fmax=10,npts=300});

[Gm,Pm] = margin(H)

Gm (a pdm) =

- domain  |
----------+----------
0.595514  | 44.5062

Pm (a pdm) =

domain    |
----------+----------
0.0257558 | 12.3814
```

Note `margin( )` also returns the frequencies at which the phase crosses the –180° line and the gain crosses the 0 dB line. These results match the gain and phase margins shown graphically in Figure 5-4.

nichols( )

```
[H,dB,Phase] = nichols(Sys,{F,keywords})
```

The `nichols( )` function is another useful frequency domain tool for examining system performance in dynamic systems. The open-loop frequency response is calculated and plotted against the gain in the standard Nichols format (gain in decibels versus phase in degrees). Different points on the plot thus correspond to different values of ω.

`nichols( )` plots are particularly useful as a means of obtaining the closed-loop frequency response of a system from the open-loop response. Nichols plots are frequently augmented with curves, or loci, of constant magnitude or phase. These curves are drawn when the `pattern` keyword is specified. Notice that each point on the open-loop response curve corresponds to the response of the system at a given frequency, and the closed-loop magnitude response at that frequency can be read off the Nichols plot by noting the value of the magnitude locus which the point on the curve intersects. The closed-loop phase can be determined in a similar manner by noting the phase locus which the open-loop curve crosses. [Oga70]

`nichols( )` is implemented in a manner very similar to that used for `bode( )`. It generates a frequency range if none is explicitly entered, calls `freq( )` internally, and converts the complex frequency response to magnitude gain in decibels and phase in degrees. `bode( )` and `nichols( )` differ only in the plots they produce. For MIMO systems, `nichols( )` will produce plots with as many curves as there are system inputs. A menu presents a selection of output responses. To generate a Nichols plot, use the syntax shown in Example 5-5.

Example 5-5 nichols( ) Plot

```
A = [2, 0, -0.01; 2,-2,0; -1.4, 3, 0];
B = [3; 5; -1];
C = [1, 0, 4];
nsys = system(A,B,C,1);
H=nichols(nsys,
   {Fmin=.01,Fmax=5,npts=300, pattern,!wrap})
```

The result is shown in Figure 5-5.

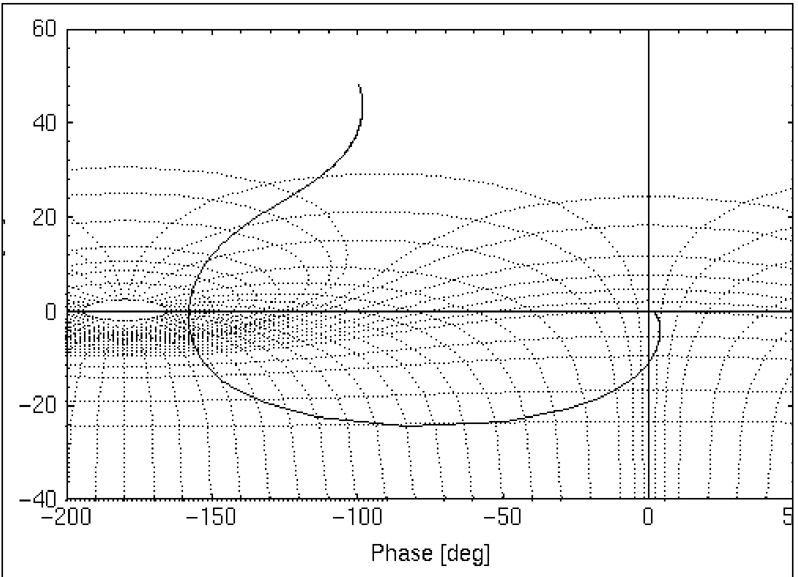


Figure 5-5. nichols( ) Gain-Phase Plot

Nyquist Stability Analysis

Nyquist analysis is a frequency domain method for examining system performance of dynamic systems. Nyquist plots typically consist of the real part of the frequency response plotted against the imaginary part of the response. Nyquist plots are particularly useful in that they indicate the stability of a closed-loop system, given an open-loop system which includes a gain, *K* (it may be unity).

Nyquist's stability criterion derives from Cauchy's principle, which states that a contour integral of a complex function will evaluate to zero as long as the contour does not contain a singularity of that function [ChB84]. The frequency response is the complex function in this case, and the contour over which it is evaluated and plotted is determined by the frequency range of the response.

Nyquist's stability criterion states that the number of clockwise encirclements of the –1 point on the real axis by the plot is equal to the number of unstable closed-loop poles minus the number of unstable open-loop poles. This criterion can be used to determine how many encirclements are required for closed-loop stability. For example, if the

plant is open-loop stable, then there should be no encirclements. If the plant has one open-loop unstable pole, there should be one negative (counter-clockwise) encirclement.

The stability criterion is most easily derived from the SISO transfer-function representation of a system. The Nyquist plot for a MIMO system consists of a set of plots, one for each output, each containing as many input frequency response curves as there are system inputs. You can derive any plot from a context menu. If you close a feedback loop around a SISO system in transfer function format, you obtain a closed-loop system as shown in Figure 5-6.

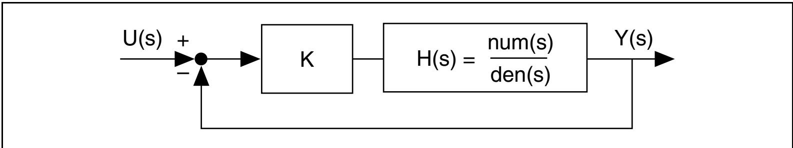


Figure 5-6. Closed-Loop System Containing a Variable Gain *K*

You obtain the following closed-loop transfer function from $Y(s)$ to $U(s)$:

$$\frac{Y(s)}{U(s)} = \frac{KH(s)}{1 + KH(s)}$$

Thus, the closed-loop roots are the roots of the equation $1 + KH(s) = 0$. The complex frequency response of $KH(s)$, evaluated for $s = j\omega$ in continuous time and $e^{j\omega T}$ for discrete systems, will encircle (–1,0) in the complex plane if $1 + KH(s)$ encircles (0,0). If you are examining the Nyquist plot of $H(s)$, you will notice that an encirclement of $(-1/K,0)$ by $H(s)$ is the same as an encirclement of (–1,0) by $KH(s)$. This fact allows you to use one Nyquist plot to determine the stability of a system for any and all values of K.

nyquist( )

```
H = nyquist(Sys,{F,keywords})
```

The `nyquist( )` function is structured very similarly to `bode( )` and `nichols( )` in that it is largely a wrapper on the `freq( )` function to obtain the system's frequency response. The output `H` is just the output from the call to `freq( )`. The main difference from the other two functions is that `nyquist( )` does not calculate the decibel gain and the phase of the system's response. It generates the Nyquist plot by plotting the real part of each point of the response against the imaginary part.

The Nyquist plot Xmath generates is complete only for the frequencies you specify. Ideally you would obtain a plot based on the frequency response from $\omega = 0$ to $\omega = \infty$. However, a good choice of frequency range usually comes close enough. When you have obtained the Nyquist plot from approximately $w = 0$ to ∞, you can reflect it about the real axis to get a complete plot of the open-loop frequency response from $-\infty$ to $+\infty$. Extend the resulting curve, traveling clockwise, until the contour is closed. Refer the augmented plots in Example 5-6. When you have done this, you can use the expression $Z = N + P$ to find the number of unstable closed-loop system roots, Z, given the number of clockwise encirclements of the $(-1/K,0)$ or $(-1,0)$ point and the number of unstable (right-half plane) poles of the open-loop system.

For an example of how to use Nyquist plots to determine stable gains for the closed-loop system, refer to Example 5-6.

Example 5-6 Using Nyquist Plots to Determine Stable Gains for the Closed-Loop System

By examining the Nyquist plot for your open-loop system

$$G(s) = \frac{(s + 0.5)}{s^2(s + 2)(s + 10)}$$

you can tell for what multiplicative gain values K the closed-loop system will be unstable.

```
H = nyquist(sys,{Fmin=0.01,Fmax=10,npts=300});
```

gives you an overview of the Nyquist plot for a broad range of frequencies, but the plot gives more information than you need about the low frequency response and not enough about the response at higher frequencies. Refer to Figure 5-7.

You do another Nyquist plot, this time examining the high-frequency response more closely. Refer to Figure 5-8.

```
H2= nyquist(sys,{Fmin=.5,Fmax=5,npts = 150})
```

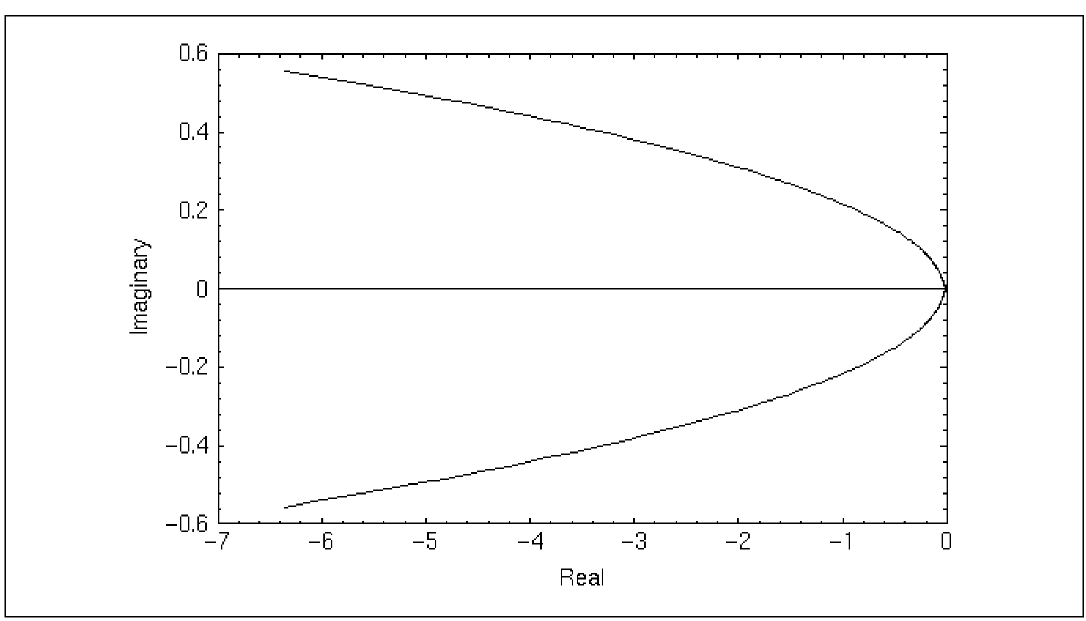


Figure 5-7. Nyquist Plot of the Open-Loop System for Frequencies from 0.01 Hz to 10 Hz

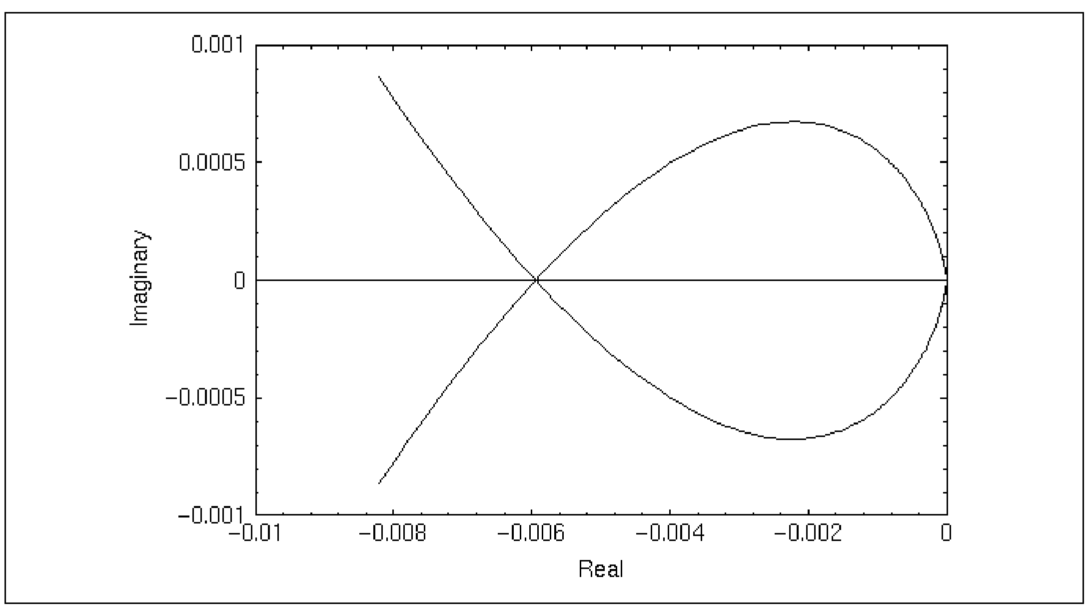


Figure 5-8. Nyquist Plot of the Open-Loop System for Frequencies from 0.5 Hz to 5 Hz

By combining the information from the two plots, reflecting them across the real axis to account for the negative frequency response and augmenting them with a line closing the contour in the clockwise direction, you obtain the sketch of the encirclement pattern shown in Figure 5-9. In this figure, Nyquist contour is formed by drawing the system's Nyquist plot for all positive frequencies, reflecting it about the real axis to show plot for negative frequencies, and completing the closed contour in a clockwise direction.

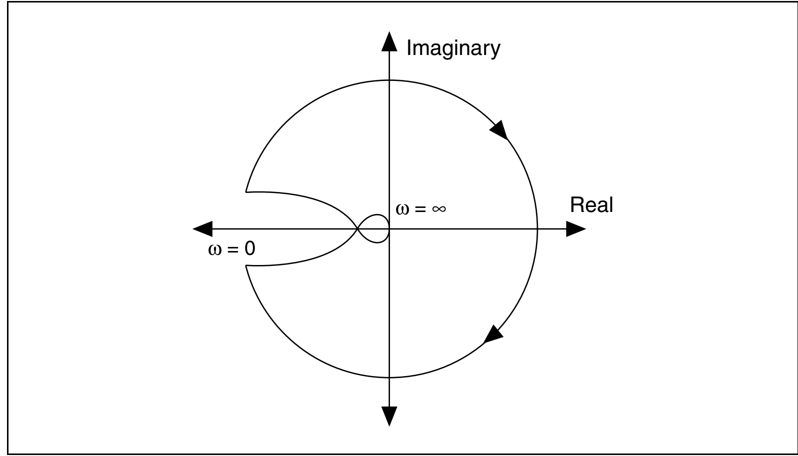


Figure 5-9. Nyquist Contour Formed by Drawing the System's Nyquist Plot for All Positive Frequencies

Because your open-loop system has no unstable (right-half plane) poles, the number of unstable closed-loop poles for a given gain K will be equal to the number of times the contour encircles the $(-1/K, 0)$ point.

Referring back to the Nyquist plots, you see that for $K > 168$ (or $-1/K > -0.006$), you have two encirclements of the $(-1/K, 0)$ point, and thus two unstable closed-loop poles. For gain values less than 167, the closed-loop system is stable. You can verify this with a small experiment, using a value of 169 for K:

```
sysu = 169*sys;
sysucl = feedback(sysu);
poles(sysucl)
```

```
ans (a column vector) =
```

```
-0.52263
0.00336213 + 3.75217 j
0.00336213 - 3.75217 j
-11.4841
```

Two of the poles of the closed-loop system are now unstable.

Linear Systems and Power Spectral Density

A key characteristic of the linear, time-invariant systems represented in Xmath is that the transfer function between a system input and a system output is just the Fourier transform of the response at that output to a delta impulse at that input. The power spectral density of a time series is defined as the Fourier transform of the autocorrelation function of the series.

Given these two concepts, you can obtain the power spectral density of the output of a linear, time-invariant system just by knowing the power spectral density of the input and the system's transfer function [Leo89], [GrD86]. Representing the transfer function by $H(q)$ and the power spectral densities of the input and output as $S_U(q)$ and $S_Y(q)$, respectively:

$$S_Y(q) = |H(q)|^2 S_U(q)$$

You also can obtain the cross-power spectral densities:

$$S_{YU}(q) = H(q)S_U(q)$$
$$S_{UY}(q) = H^*(q)S_U(q)$$

These results indicate that you can shape the spectrum of a linear system's output by using an input with an appropriate spectrum. Alternatively, you can choose a system to give you the output spectrum you want, given a fixed set of input data. When you use linear systems in transfer-function form for such applications, you generally refer to them as filters rather than systems.

psd( )

```
[Ypsd,Yspec] = psd(Sys,{Uspec})
```

The `psd( )` function computes the power spectral density and cross-spectral density of a system's outputs as a function of frequency, given the frequency-dependent input power spectral density matrices. The input parameter `Uspec` is a PDM where domain contains the frequency

values at which the power spectral density is to be computed and where dependent matrices are the input power spectral density matrix at each frequency. `psd( )` computes a cross power spectral density matrix for each of a user–specified set of frequency values ω, returning them together in the PDM `Yspec`:

$$Y_{spec} = H \times (jw) \times U_{spec} jw \times H(-jw)$$

`psd( )` calls `freq( )` internally to compute the frequency response, *H*, of the system. It uses the frequency range specified by the domain of `Uspec`.

The power spectral density of the output as a function of frequency, given in `Ypsd`, is obtained from the real parts of the diagonal terms of the dependent matrices in `Yspec`.

Some background information on power spectral density may be useful. Given a time-domain input series $U(t)$, the power spectral density of $U(t)$ is the Fourier transform of the autocorrelation of $U(t)$. For a system with q inputs each input spectral density dependent matrix within `Uspec` is a square Hermitian matrix of size q. A Hermitian matrix is a square matrix equal to its complex conjugate transpose. If `Uspec` is constant for all frequencies (when the spectrum is white) then `Uspec` can be specified as a single matrix.

If you are working with multiple systems which have been cascaded in series, the output power spectral density of the first system can be used as the input power density to the second system in a subsequent use of `psd( )`.

For an example of how to verify the response of a system to white noise input, refer to Example 5-7.

Example 5-7 Verifying the Response of a System to White Noise Input

You can easily generate the power spectral density of an input white noise process.

```
sys = polynomial(-0.5)/polynomial([0,0,-2,-10]);
```

```
w = logspace(0.01,1,50);
Uspec = pdm(ones(w),w);
```

You then use `psd( )` to obtain the output power spectral density and the cross-spectral density as a function of frequency.

```
[Ypsd,Yspec] = psd(sys,Uspec);
```

6

State-Space Design

The functions in this chapter are generally termed “modern control” tools. They are based on the state-space linear system representation, and employ methods which are generally applicable to both SISO and MIMO problems. For a review of the state-space system representation, refer to the *State-Space System Models* section of Chapter 2, *Linear System Representation*.

The process of state-space control system design comprises several distinct steps. First, you need to assess the controllability and observability of the system. The designs discussed in this chapter are based on systems that are both controllable and observable. When you have determined the controllability and observability of the system, you can design a feedback control law based on the set of state values. Next, you design an estimator that estimates the state variable values based on the measured output. Finally, you combine the controller and estimator to obtain a complete compensator for the system.

In designing optimal control systems, you pick a performance index you want to optimize for a given system. This performance index is a quadratic function reflecting the physical constraints of the system and the characteristics of any noise that may be present. When this performance index is a quadratic, you solve mathematically for the optimal control law and estimator as discussed in the *Linear Quadratic Regulator* section and the *Linear Quadratic Estimator* section.

This chapter concludes with a discussion of system balancing. The controllability and observability grammians provide a measure of how controllable and observable a system is. They also can be used to transform a system to its internally balanced form.

Controllability

Controllability is the property of being able to move the states of a system arbitrarily in a finite time, given some control input to the system. Although a particular physical system may be controllable by this definition, not all state-space models describing that system may be controllable. For example, if there exists a system eigenvector orthogonal to the input

matrix B, then the mode of the system associated with the corresponding eigenvalue cannot be controlled with any input. You can think of this in the SISO transfer function case as a cancellation between a numerator and denominator root—where you cannot control the system in the direction of that root (mode).

It can be shown (refer to [Kai80]) that for a continuous-time system with the state update equation:

$$\dot{x} = Ax + Bu \tag{6-1}$$

you can define the controllability matrix for both continuous and discrete systems as:

$$C = [B \;\; AB \;\; \mathrm{A}^2 B \ldots A^{n-1} B] \tag{6-2}$$

For all modes of the system to be controllable, the controllability matrix C must contain a linearly independent column vectors for each system mode. Thus, with A an $n \times n$ matrix, C must have rank n for the system to be controllable.

In the context of gain-state feedback, a system's controllability determines whether you may be able to change the effective dynamics of the system to ones that yield a more desirable response.

Using full-state feedback, as shown in Figure 6-1, so that $u = v - Kx$. Working through the system equations, you obtain

$$\dot{x} = (A - BK)x + Bv \tag{6-3}$$

for the new state-update equation. If the system is controllable, you can relocate the eigenvalues of the closed-loop system to any value by choosing the vector of state gains K appropriately. Conversely, the eigenvalues associated with uncontrollable modes remain unchanged, no matter what value you choose for K.

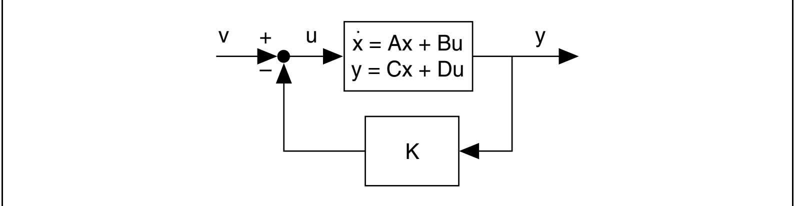


Figure 6-1. Full-State Feedback Being Used to Relocate the Eigenvalues of a Controllable System Based on the Value of the Gain *K*

controllable( )

```
[SysC,T,nuc]=controllable(Sys,{tol})
```

The question that naturally arises is, "How do you know which states are controllable in a given system?" The `controllable( )` function returns the controllable partition of a state-space system, the number of uncontrollable states in the original system, and a linear transformation matrix which can be used to partition the states into controllable and uncontrollable sets. For an example of how this is done, refer to Example 6-1.

`controllable( )` uses the staircase algorithm, which is discussed in more detail in the *stair( )* section.

Example 6-1 Controllability of a System

Perform `controllable( )` on a system is described by:

```
A = [1,0,0.01;0,1,0;0,0,1];
B = [1,0,0]'; C = [0.6,0.8,0];D = 0;
Sys = system(A,B,C,D);

[SysC,T,nuc] = controllable(Sys)

The system has 2 uncontrollable states

SysC (a state space system) =

A
1

B
-1

 C
-0.6

D
0
```

```
X0
0

Input Names
-----------
Input 1

Output Names
------------
Output 1
System is continuous

T (a square matrix) =

     2.22045e-16   0      -1
     0             1       0
    -1             0       2.22045e-16

nuc (a scalar) = 2
```

These results indicate that only the first state of the system corresponds to a controllable mode, and the remaining two are uncontrollable.

Similarly, if you form the controllability matrix for this system,

```
[,states] = size(A);
Con = B;
For i = 1:states-1;
   Con = [B, A*Con];
endFor
det(Con)

ans (a scalar) = 0
```

you see that the controllability matrix is singular (its determinant is zero), confirming the results from `controllable( )`.

Observability and Estimation

As described in the *Controllability* section, the term *controllability* describes whether or not a system's states can be affected, and the system eigenvalues relocated, by changes to the system input. The analogous concept of observability describes whether it is possible to determine the value of an individual state at a particular time by observing the system outputs for a finite amount of time. In essence, an observable system is one for which you can "observe" state values by knowing the output of the system.

Beginning with the basic state-space equations (the *Du* output term can be omitted without loss of generality):

$$\dot{x} = Ax + Bu$$
$$y = Cx$$

you can obtain expressions for the successive derivatives of the output term, thus forming a complete description of the initial condition on the output:

$$\begin{bmatrix} y \\ \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} C \\ CA \\ CA^2 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 0 \\ CB & 0 & 0 \\ CAB & CB & 0 \end{bmatrix} \begin{bmatrix} u \\ \dot{u} \\ \ddot{u} \end{bmatrix}$$

Generally, you term the following matrix

$$O = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

the observability matrix. If the rank of O is equal to n (where A is an $n \times n$ matrix), you can always find a state vector which will realize the initial conditions you want on the output, presuming that you know the initial conditions on the input. If, however, the observability matrix loses rank (is singular, in the SISO case), you will not be able to find states that give you particular output conditions if those conditions lie in the null space of the observability matrix.

The observability of a system is of particular importance when you want to determine the actual values of the states based on our knowledge of the system dynamics and the input and output values at a given time. If a system is observable, you can create an observer, or estimator, to “guess” the values of the states and use the information available to zero the state estimate error as quickly and accurately as possible.

Referring to Figure 6-2 and tracing through the system equations, you can obtain the time-update equation for the state estimate error $\tilde{x} = x - \hat{x}$. Figure 6-2 is a general observer block diagram where the output estimate error $\tilde{y}$ is defined as $\tilde{y} = y - \hat{y}$.

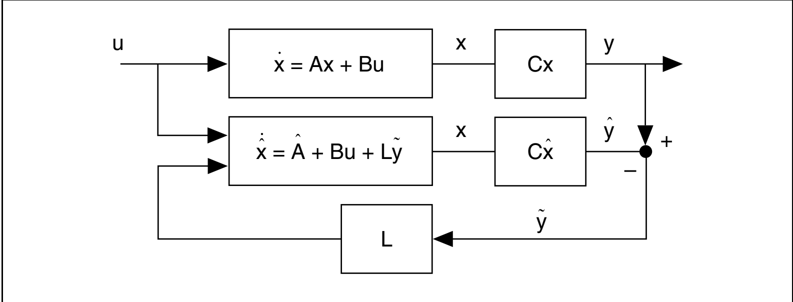


Figure 6-2. General Observer Block Diagram

If the observability matrix is nonsingular, you will be able to put the eigenvalues (pole locations) of $(A - LC)$, shown in Equation 6-4, anywhere you want. Thus, you can choose them to make $\tilde{x}$ decay to zero as quickly as possible.

$$\dot{\tilde{x}} = (A - LC)\tilde{x} \tag{6-4}$$

The problem of finding the eigenvalues of $(A - LC)$ can be equivalently posed as that of finding the eigenvalues of $(A' - C'L')$. This statement can be recognized as equivalent to that of the pole-placement problem for a state-feedback controller (refer to the new state-update equation in the *Controllability* section), with A, B, and K replaced by A', C', and L', respectively. Notice that these two representations correspond to a state-space system and its transpose. This illustrates the principle of duality between the controller and estimator forms. For more information, refer to the *Duality and Pole Placement* section.

observable( )

```
[SysO,T,nuo] = observable(Sys,{tol})
```

The `observable( )` function is the analogue to `controllable( )`. As described in the *Controllability* section, if a system {A,B,C,D} is controllable, its transpose {A',C',B',D'} is observable. `observable( )` returns the observable partition of a state-space system, the number of unobservable states in the original system, and a linear transformation matrix which can be used to partition the states into observable and unobservable sets. For an example of how to use the `observable( )` function, refer to Example 6-2.

`observable( )` uses the staircase algorithm, which is described in more detail in the *stair( )* section.

Example 6-2 Observability of a System

A system is described by:

```
A = [1,0,0.01;0,1,0;0,0,1];
B = [1,0,0]';C = [0.6,0.8,0];D = 0;
Sys = system(A,B,C,D);
```

Performing,

```
[SysO,T,nuo] = observable(Sys);
```

```
The system has 1 unobservable state
```

This example indicates that one state of the system's states corresponds to an unobservable mode, but that the other two are observable.

Similarly, if you form the observability matrix for this system,

```
[,states] = size(A);
Obs = C;
For i = 1:states-1;
   Obs = [C; Obs*A];
endFor
det(Obs)
```

```
ans (a scalar) = 0
```

you see that the observability matrix is singular (its determinant is zero), confirming the results you saw from `observable( )`.

Minimal Realizations

All state-space systems have an infinite number of realizations. All systems have a minimum number of states needed to express the system dynamics, but can be described using any number of states greater than or equal to this minimum number. If a system has more states than are needed to express a given transfer function, it will have unobservable and/or uncontrollable modes corresponding to eigenvalues of the A matrix that are not poles of the transfer function.

All minimal realizations of the same system are related by a coordinate transformation.

minimal( )

```
[SysM,T,nuco] = minimal(Sys,{tol})
```

Because nonminimal systems are uncontrollable, unobservable, or both, you want to be able to compute the minimal realization for a given system. This comprises the controllable and observable parts of the dynamic system. The `minimal( )` function calls both the `controllable( )` and `observable( )` functions, then extracts the part of the original system that is both controllable and observable.

`minimal( )` is implemented directly as a wrapper on `controllable( )` and `observable( )`. The controllable subsystem is extracted first, then the observable part of the subsystem is returned. For an example of how to find a minimal realization for a system with uncontrollable or unobservable parts, refer to Example 6-3.

Example 6-3 Finding a Minimal Realization for a System

A system is described by:

```
A = [1,0,0.01;0,1,0;0,0,1];
B = [1,0,0]';
C = [0.6,0.8,0];
D = 0;
Sys = system(A,B,C,D);
```

Notice that the system has a number of zero-pole pairs which cancel each other out:

```
poles(Sys)

ans (a column vector) =

    1
    1
    1
zeros(Sys)

ans (a column vector) =

    1
    1
```

To find the minimal part of the system:

```
SysM = minimal(Sys);

The system has 2 uncontrollable states

poles(SysM)
```

```
ans (a scalar) = 1
zeros(SysM)
ans is null
```

stair( )

```
[SysT,T,nc] = stair(Sys,tol)
```

The `stair( )` converts a dynamic system to staircase form. In the staircase form, the A and B system matrices are linearly transformed so that they are partitioned into controllable and uncontrollable parts. By duality, converting the transpose of a system into staircase form results in its being separated into observable and unobservable parts. The matrices are partitioned as shown in the following example:

$$A_{stair} = \begin{vmatrix} A_{uc} & 0 \\ A_{cuc} & A_c \end{vmatrix} \quad \mathrm{B}_{stair} = \begin{vmatrix} 0 \\ B_c \end{vmatrix}$$

where A_c is controllable, A_{uc} is uncontrollable, and A_{cuc} represents the coupling from the uncontrollable states to the controllable part of the representation. There is no coupling from the controllable states to the uncontrollable ones.

T is a transformation matrix between the original system, `Sys`, and the system in staircase form, `SysT`.

The transformations are as follows:

$$A_{stair} = T^{-1}AT$$

$$B_{stair} = T^{-1}B$$

$$C_{stair} = CT$$

The optional tolerance, `tol`, indicates the threshold value beneath which numbers in the transformed matrices should be rounded to zero.

The staircase algorithm used to partition the system derives from the Van Dooren algorithms. For further details, refer to [Van79], [Van81], and [BeV88].

Duality and Pole Placement

The new state-update equation in the *Controllability* section and the *Observability and Estimation* section, the time-update equation in the *Observability and Estimation* section, along with the corresponding block diagrams in Figures 6-1 and 6-2, indicate how you can move the eigenvalues, or poles, of a minimal system through the choice of a feedback gain *K* (or *L*). Given the system's state-space representation and any desired set of closed-loop poles, you can solve an eigenvalue problem to find the gain that yields these poles for the complete system. Although the poles of a minimal system can be moved to any value, this approach does not guarantee that the resulting gain is small or physically practical—just that it is finite.

The similarity between the new state-update equation (Equation 6-1) and the time-update equation (Equation 6-4) for controller and observer feedback brings up the principle of duality with respect to the controllability and observability of a system. Briefly, for a given state-space system Sys_1 with system matrices {A,B,C,D}, there exists a dual system Sys_2 described by {A*,C*,B*,D}, using * to denote a complex conjugate transpose. If Sys_1 is controllable, Sys_2 will be observable, and vice versa. This can be quickly verified by constructing the controllability and observability matrices for both. Thus, the gain value that yields a set of desired closed-loop poles for feedback control of a system also yields an observer with the same pole locations for the system's dual.

poleplace( )

```
K = poleplace(A,B,poles)
```

The `poleplace( )` function solves the problem `eig(A – B * K) = poles` for single-input systems. This is essentially the problem posed in Figure 6-1 and the new state-update equation in the *Controllability* section. If you know where you want the system poles to be located, `poleplace( )` returns the value of the gain vector `K` that will move the closed-loop poles to the desired locations.

The syntax `poleplace(A',C',poles)` can be used for regulator problems, or by duality, for estimator problems. In general, the system {A,B} must be reachable (all unstable poles controllable) for controller design and the system {A',C'} must be stabilizable (all unstable poles observable) for estimator design. The current `poleplace( )` implementation is limited to single-input systems.

`poleplace( )` is unusual among Xmath's modern control design functions in that only the A and B matrix variables are used as input, rather than a complete system variable. This is done because the other state-space matrices are not needed in the computation, and in many cases it is desirable to change or perturb the elements of the A and B matrices slightly to simulate actual conditions, without having to reformat the entire system. For an example of an arbitrary pole placement for a controllable system, refer to Example 6-4.

Example 6-4 Arbitrary Pole Placement for a Controllable System

```
A = [0,1,0,0;21,0,0,0.8;0,0,0,1;0,0,0,-4];
B = [0,-2,0,1]';
C = eye(4,4);
D = zeros(4,1);
ipsys = system(A,B,C,D);
```

If you want to place the system poles in a Butterworth pattern:

```
Kc = poleplace(A,B,[-5+8.66*jay, -8.66+5*jay]);
```

You then can use this new gain vector as a feedback gain to create a new system,

```
ipsysfb = feedback(ipsys, system([],[],[],Kc));
```

and verify that the poles of this new system are at the designated locations:

```
poles(ipsysfb)

ans (a column vector) =

-8.66 + 5 j
-8.66 - 5 j
-5 + 8.66 j
-5 - 8.66 j
```

You need specify only one complex pole in a conjugate pair of desired pole locations. `poleplace( )` checks for conjugate pairs and adds conjugates as necessary to the input `poles( )` vector.

Then the system matrix S is created:

$$S = \begin{bmatrix} A & B \\ r & 0 \end{bmatrix}$$

r is a random row vector with as many rows as A has columns. You then create a random complex row vector with as many elements and conjugate

pairs as `poles( )`. For each pole value in `poles( )`, `poleplace( )` forms a vector by subtracting the pole's value from each diagonal element of *S* except for the last element (0). The resulting matrix is then divided by the corresponding value in the random complex vector. The complex value is padded with zeros to form a vector that is row compatible with the matrix. `poleplace( )` then divides the last element of this quotient vector by the negative of the first element of the quotient vector, and the result is the gain required to move that pole value. This sequence of steps is performed as a matrix operation so that the complete gain vector is computed immediately. `rcond( )` is called to examine the condition of the matrix formed by all the quotient vectors. If the condition number returned is less than `eps` × (the row size of A), `poleplace( )` displays a warning message indicating that the eigenvectors of the closed-loop system are ill-conditioned.

Linear Quadratic Regulator

A regulator is a feedback controller designed to drive the states of a controllable system using acceptable amounts of control and keeping the states within acceptable levels (where the designer can mathematically define what constitutes "acceptable" in both cases). Figure 6-3 shows a continuous-time regulator where the design presumes availability of all states, feeding them back through the optimal gain array K_r to drive the system so that the states return to zero as quickly as possible in the presence of a disturbance or noise, represented by ω.

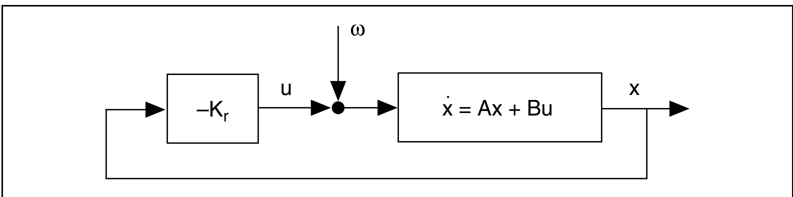


Figure 6-3. Continuous-Time Regulator

In designing a regulator, the goal is to find a controller that minimizes the effects of disturbances on the states of the system. Xmath's linear quadratic regulator function, `regulator( )`, uses a quadratic performance index to establish the trade-off between the permissible state fluctuation and the available energy, or amount of control, required to move the states.

Note In designing a regulator, assume that all the states of the system are available as outputs.

For continuous-time systems,

$$\dot{x} = Ax + Bu$$

the quadratic performance index takes the form:

$$J = \int_0^{\infty} \begin{bmatrix} x'(t) & u'(t) \end{bmatrix} \begin{bmatrix} R_{xx} & R_{xu} \\ R_{xu'} & R_{uu} \end{bmatrix} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt$$

For the discrete case where the system is defined as a multistage process:

$$x_{k+1} = Ax_k + Bu_k$$

the performance index is defined similarly except that a summation sign replaces the integral of the preceding quadratic performance index equation.

R_{xx} is a real, symmetric, positive-semidefinite matrix indicating the weighting of the cost on the elements of the state vector *x*. R_{uu} is a real, symmetric, positive-definite matrix indicating the weighting of the cost on the control inputs given by the vector *u*. R_{xu} is a real matrix indicating the cross-weighting of the cost between states and inputs; for many applications, it will consist of all zeros if the control and states are uncorrelated.

Bryson and Ho showed in [BH75] that the optimal control which minimized this quadratic performance index is a linear feedback combination of the states, $u = K_r x$, for both the continuous and discrete cases.

For the continuous case, K_r is defined as follows, with *P* solving the continuous-time Riccati equation:

$$K_r = R_{uu}^{-1}(B'P + R_{xu})$$

$$R_{xx} + PA + A'P - (PB + R_{xu})R_{uu}^{-1}(R_{xu'} + B'P) = 0$$

and for the discrete case, *P* solves the discrete Riccati equation.

$$K_r = (R_{uu} + B'PB)^{-1}(B'PA + R_{xu})$$

$$A'PA - (A'PB + R_{xu})(R_{uu} + B'PB)^{-1}(B'PA + R'_{xu}) + R_{xx} = P$$

The optimal estimator and regulator problems illustrate the principle of duality—that for any given system realization {A,B,C} there is a dual realization {A',C',B'} with related controllability and observability. Refer to the *Duality and Pole Placement* section.

regulator( )

```
[Kr,ev,P] = regulator(Sys,Rxx,Ruu,{Rxu})
```

The `regulator( )` function calculates the optimal gain matrix K_r for a given dynamic system with specified state weighting, control weighting, and (optionally) cross-weighting matrices.

Alternatively, K_r can be obtained through a call to `riccati( )`:

```
[P,resid,Kr,ev]=riccati(Sys,Rxx,Ruu,{S=Rxu})
```

The syntax for `riccati( )` is discussed in the *Riccati Equation* section.

As shown in the diagram of a continuous-time regulator in Figure 6-3, the state equation for the regulator is the following:

$$\dot{x} = (A - BK)_r x$$

If you want the closed-loop system eigenvalues, compute them as the eigenvalues of $(A - BK_r)$.

If numerical difficulties are encountered, the algorithm will attempt to determine whether or not the problem is well posed. Checks are made to determine stabilizability and the positive definiteness or semipositive-definiteness of the cost functionals.

The most important design parameters are R_{xx} and R_{uu}, which need to be chosen to reflect the real limitations on how much control can be provided, or how problematic large state values can be. For an example of how to design a regulator for the inverted pendulum, refer to Example 6-5.

Example 6-5 Designing a Regulator for the Inverted Pendulum

A classic control design problem, the inverted pendulum, consists of a rod (the pendulum) hinged to the top of a cart which can be moved freely in either direction along a line. The goal of the controller is to supply an input u such that the pendulum will be maintained in a vertical position ($\phi = 0$, in Figure 6-4).

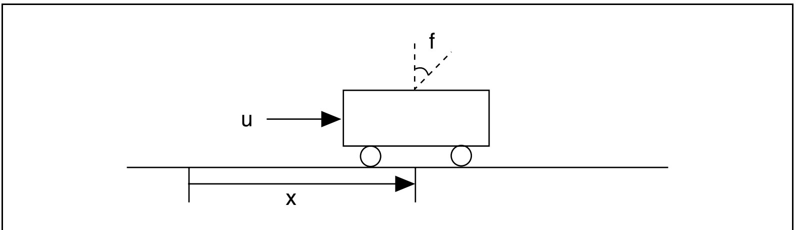


Figure 6-4. Diagram of Plant for the Inverted Pendulum Problem

Figure 6-4 shows the pendulum at $\phi = 0$ and $\phi > 0$. The distance of the cart from some initial reference point along the line of its motion is represented by the state variable *x*. You can measure the angle ϕ and the distance *x* easily—in fact, you will use measurements as your system outputs—but it is more difficult to obtain accurate measurements of the rate at which *x* and ϕ change.

Designating $[\phi\ \dot{\phi}\ x\ \dot{x}]'$ as the state vector, you can set up the system in Xmath:

```
A = [0,1,0,0;21,0,0,0.8;0,0,0,1;0,0,0,-4];
B = [0,-2,0,1]';
C = [1,0,0,0;0,0,1,0];
D = [0,0]';
ipsys = system(A,B,C,D);
```

You design a regulator with the assumption that all four states are available. Recalling that you defined the state vector as $[\phi\ \dot{\phi}\ x\ \dot{x}]'$, you can decide the weighting you want to associate with each state. Refer to the quadratic performance index equation in the *Linear Quadratic Regulator* section for more information. For this particular problem, your most important performance goal is that the pendulum stay upright—that is, that ϕ be tightly controlled to stay as close to zero as possible. You also might prefer, though to a lesser extent, that you not have to move the cart over too great a distance. Physical limitations, such as the size of the room in which the experiment is conducted, should be considered.

If you are not particularly concerned about the speed of the cart across the floor or that rate of change of the angle, you might define,

```
Rxx = diagonal([1,0,0.1,0]);
```

with the larger values in the matrix corresponding to states whose values you care most about. Presuming you are not too worried about the size of the input *u* you impart to the cart:

```
Ruu = 1e-5;
```

`Ruu` is a scalar because you have only one input for this particular model.

```
[Kr,ev,P] = regulator(ipsys,Rxx,Ruu);
Kr

Kr (a row vector) =

-348.778  -32.1056  -100  -27.3036
```

Note You will use this regulator gain later in designing a compensator.

Linear Quadratic Estimator

The LQR approach discussed in the preceding section is based on the assumption that the values of all the states are available. In the real world, only the output values are generally available and they are frequently corrupted with noise. You know from the *Observability and Estimation* section that you can obtain an estimate of the states using an observer if the system is reachable. The problem solved with the optimal estimator function `estimator( )` is that of finding the best estimate of the states, given certain assumptions about the noise associated with the output.

As shown for the continuous case in Figure 6-5, the plant system is augmented with an estimator—an observer used in conjunction with a noisy system. The estimator supplies estimates of all the system states and feeds back the difference between the estimated and the actual outputs through the optimal estimator gain K_e. `estimator( )` calculates the constant, optimal state-estimator gain matrix K_e for a dynamic system. The estimator gain is derived by minimizing the expected mean square of the error between the measured output y and the output from the estimator, $\hat{y}$. This model takes into account that there may be some process noise within the system model (plant) itself as well as some noise inherent in the device used to measure the outputs.

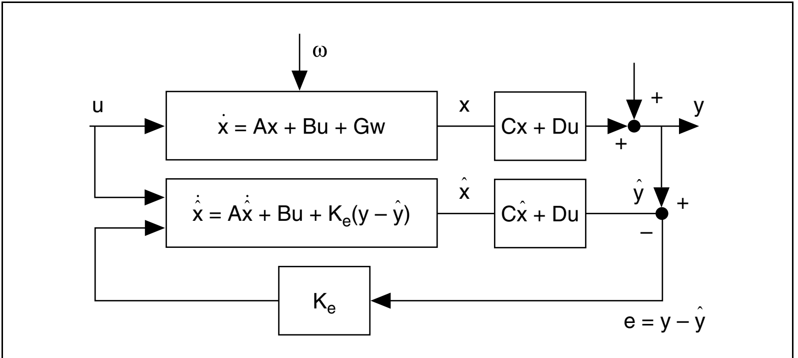


Figure 6-5. Diagram of the Estimator Representation

`estimator( )` inputs include the dynamic system `Sys`, and the noise intensity matrices Q_{xx}, Q_{yy}, or Q_{xy}. For a linear–time–invariant process described by:

```
Sys = system(A,B,C,D)
```

The following equation describes the complete plant:

$$\dot{x} = Ax + Bu + Gw$$
$$y = Cx + Du + n$$

A, *B*, *C*, and *D* are directly from the previous state-space system representation where ω is the input disturbance, *G* is the input disturbance matrix and ν is the measurement noise. The noise intensity matrices are defined as,

$$E(v(t)v'(\tau)) = Q_{yy}\delta(t-\tau)$$
$$E(G\omega(t)\omega'(\tau)G') = Q_{xx}\delta(t-\tau)$$
$$E(G\omega(t)v'(\tau)) = Q_{xy}\delta(t-\tau)$$

where *E* is the expectation operator and δ is the delta function.

The noises ω and ν are assumed to be white and zero mean. Q_{yy} has matrix dimensions equal to the number of plant outputs and must be positive definite, while Q_{xx} has matrix dimensions equal to the number of plant states and must be positive semi-definite. In many cases the input disturbances and output noises are uncorrelated so that $Q_{xy} = 0$. If

numerical difficulties are encountered, the algorithm will attempt to determine whether or not the problem is well posed. Checks are made to determine the reachability and the positive definiteness or semipositive-definiteness of the covariance matrices.

Because not all the values in the state vector are directly available from measurements, your goal is to find an estimate of the state vector which minimizes, in a least-squares sense, the error between the actual state vector and the estimated state vector. This estimated vector is denoted by $\hat{x}$. Because you want to minimize the error between this estimate and the actual state values, the quadratic expression to be minimized becomes:

$$J = \int_0 \left[(x(t) - \hat{x}(t))' \;\; (y(t) - \hat{y}(t))' \right] \bullet$$

$$\begin{bmatrix} Q_{xx} & Q_{xy} \\ Q_{xy'} & Q_{yy} \end{bmatrix} \begin{bmatrix} (x(t) - \hat{x}(t)) \\ (y(t) - \hat{y}(t)) \end{bmatrix} dt$$

For the case of a discrete-time system, this quadratic expression is evaluated as a summation rather than as an integral. No additional information is provided by the inclusion of the Du term, so it can be omitted without loss of generality.

A derivation of the differential equation for the continuous-time state vector estimate, $\hat{x}$, can be found in [Kai81]. In the limit, this differential equation, which provides the values for the continuous-time optimal estimator, is

$$\dot{\hat{x}} = (A - K_e C)\hat{x} + Bu + K_e y \tag{6-5}$$

where $K_e = (PC' + Q_{xy}')Q_{yy}^{-1}$ and where the matrix P is obtained by solving the algebraic Riccati differential equation:

$$0 = PA' + AP - (PC' + Q_{xy})Q_{yy}^{-1}(Q_{xy'} + CP) + Q_{xx}$$

The two preceding equations describe the continuous-time Kalman-Bucy filter [KaB61].

The discrete-time estimator follows from a similar system description, using the discrete-time difference equation representation of the system, as shown in the following equations.

$$x_{k+1} = Ax_k + Bu_k + G\omega$$

$$y_k = Cx_k + Du_k + \nu$$

You obtain the discrete-time estimator by considering the state estimate at two separate stages. Begin with the assumption that an estimate of the state exists prior to each measurement of the output information. This pre-existing estimate is called $\bar{x}_k$. The estimated state value after each measurement update is denoted by $\hat{x}_k$. This method takes into account the fact that the system's states change between measurements due to the system dynamics. The optimization problem, then, consists of minimizing the estimate error covariance M after each measurement update. This minimization is performed in [Kai81]. This problem is expressed in the same manner as in the preceding quadratic expression (for J), except that a summation sign replaces the integral as you are working with discrete data and you replace the variable J with M, to denote that this covariance follows each measurement update.

In determining the state values x_k from each measured y_{yk}, consider the time just prior to a new measurement for y_k. At this point $\bar{x}_k$ and M_k are the current estimates for the state and covariance. $\bar{x}_k$ is derived from the previous measurement $\hat{x}_{k-1}$, and M_k is derived from the previous post-measurement error covariance, P_{k-1}, as shown in Equation 6-6.

$$\begin{aligned} \bar{x}_k &= A\hat{x}_{k-1} + G_k \\ M_k &= AP_{k-1}A' + GQ_{xx}G' \end{aligned} \tag{6-6}$$

This is referred to as the time update, because you are propagating the state forward in time until the next measurement arrives.

Then, at the time immediately following the measurement, you effect the measurement update. This reflects the new information in an improved state estimate $\hat{x}_k$ and a somewhat smaller covariance, P_k. The equations for

this measurement update, derived in [Kal60], are shown in the following equations.

$$\hat{x}_k = \bar{x}_k + K_e(y_k - C\bar{x}_k)$$

$$P_k = (M_k^{-1} + C'Q_{yy}^{-1}C)^{-1}$$

Substituting the system and noise matrices for the steady-state case, you solve the discrete Riccati equation to obtain P and thence K_e, as shown in Equation 6-7 and Equation 6-8.

$$\bar{A}'P\bar{A} - \bar{A}'PC'(Q_{yy} + CPC')^{-1}CP\bar{A} + \bar{Q}_{xx} = P \qquad (6\text{-}7)$$

where

$$\bar{A} = A' - C'Q_{yy}^{-1}Q_{xy}$$

$$\bar{Q}_{xx} = Q_{xx} - Q_{xy'}Q_{yy}^{-1}Q_{xy}$$

and the discrete feedback gain K_e is given by

$$K'_e = (Q_{yy} + CPC')^{-1}(CPA' + Q_{xy}) \qquad (6\text{-}8)$$

estimator( )

```
[Ke,ev,P] = estimator(Sys,Qxx,Qyy,{Qxy})
```

The `estimator( )` function calculates the optimal gain matrix K_e for a given dynamic system with specified process, measurement, and (optionally) cross-weighting noise matrices.

Alternatively, K_e can be obtained through a call to `riccati( )`:

```
[P,resid,Ke,ev]=riccati(Sys',Qxx,Qyy,{S=Qxy})
```

The syntax for `riccati( )` is described in the *Riccati Equation* section.

As shown in the estimator diagram in Figure 6-4, the state equation for the estimator is:

$$\dot{\hat{x}} = (A - K_eC)\hat{x} + (B - D)u + G\omega$$

If you want the closed-loop system eigenvalues, compute them as the eigenvalues of $A - K_eC$. For an example of how to design a state estimator for the inverted pendulum problem, refer to Example 6-6.

Example 6-6 Designing a State Estimator for the Inverted Pendulum Problem

Most systems have some level of internal process noise that affects the value of the states. Returning to the inverted-pendulum plant of Example 6-5, assume that internal disturbances enter the system with the inputs. You thus can define a Q_{xx}, which is a function of the input matrix:

```
Qxx = 0.25*B*B';
```

Similarly, allow for some disturbance noise affecting the accuracy of the outputs you measure from the system. You have two output measurements for this system, thus two separate sources of noise. Assume that the noise affecting one output measurement does not affect that other, and that the effects of measurement noise are rather small for this instance.

```
Qyy = diagonal([1e-6,3e-6]);

[Ke,ev,P] = estimator(ipsys,Qxx,Qyy);

Ke

Ke (a rectangular matrix) =

  42.6012   -6.21395
965.35   -159.818
 -18.6419    4.67597
-401.88     68.8522
```

Now that you have access to a set of augmented states for the system (found with the differential equation for the continuous state vector shown in Equation 6-5), you can find the optimal controller based on the assumption of full-state feedback.

Linear Quadratic Gaussian Compensation

Many real-world control system design problems lend themselves to solutions using a regulator, except that not all the states are available as directly measured or computed outputs.

A compensator combines your ability to control a system using full state feedback with our ability to estimate the system states given the system output. You can design the controller and estimator separately and then combine them to make the system respond as desired, based on the measured output. The combination of system, controller, and estimator into

a compensator is shown in Figure 6-6. This figure combines full-state regulator with gain K_r and state estimator with gain K_e.

Combining the plant, or system, equations with those of the regulator and estimator, you can simplify the system equations for the compensator as follows:

$$\dot{\hat{x}} = A\hat{x} + Bu + K_e(y - (C\hat{x} + Du))$$

$$\dot{\hat{x}} = A\hat{x} - K_eC\hat{x} - (K_eD - B)u + K_ey$$

$$\hat{x} = [A - K_eC - (B - K_eD)K_r]\hat{x} + K_ey$$

$$u = -K_r\hat{x} + (0)y \tag{6-9}$$

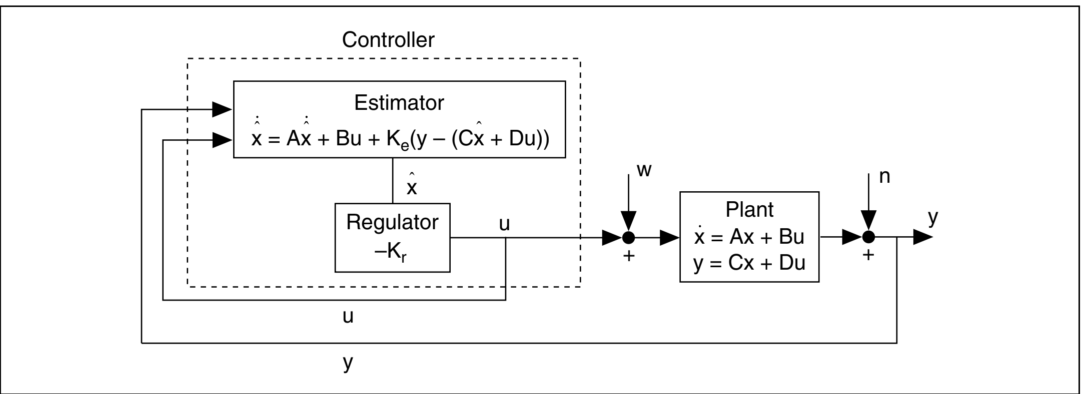


Figure 6-6. Linear Quadratic Gaussian Compensator (in the Bold Rectangle)

Equation 6-9 describes the state-space equations for both the continuous-time compensator and the discrete-time compensator if no unit delay is used between the time at which an input arrives at the system and the time at which the new output appears. However, if you are working with a real-time system which enforces a unit delay between the measurement and the control update, you will need to create a “direct” compensator in predictor form. With this direct implementation, the system output equations become the same as the state update equations, multiplied by a factor of the regulator gain K_r.

lqgcomp( )

```
SysC = lqgcomp(Sys,Kr,Ke,{direct})
```

The `lqgcomp( )` function creates a dynamic compensator given a dynamic system having at least one state and the regulator and estimator gain matrices. The returned compensator `SysC` is always in state-space form.

The regulator and estimator gains K_r and K_e need to have been calculated prior to the call to `lqgcomp( )`. You can use `regulator( )` and `estimator( )` to compute these gains if they are not already available. These functions give you the option to incorporate the presence of the white process and measurement noises ω and ν in your model, as shown in Figure 6-6.

Example 6-7 uses the inverted pendulum problem estimator and regulator gain vectors (obtained in the preceding two examples) to form a compensator. Notice that for this example you use the optional fields of the `system( )` function to store information that will be useful when you combine and simulate the compensator-plant system.

Example 6-7 Combining the Regulator and Estimator into a Full Compensator

```
A = [0,1,0,0; 21,0,0,0.8; 0,0,0,1; 0,0,0,-4];

B = [0,-2,0,1]';
C = [1,0,0,0;0,0,1,0];
D = [0;0];

ipsys = system(A,B,C,D, {inputNames = "Force (u)",
    outputNames = ["phi","x"],
    stateNames = ["phi","d(phi)","x","d(x)"]});

Kr=regulator(ipsys,diagonal([1,0,.1,0]),1e-5);

Ke=estimator(ipsys,0.25*B*B',
    diagonal([1e-6,3e-6]));

ipsysc=lqgcomp(ipsys,Kr,Ke);
```

Now that you have the compensator, you need to connect it to the original plant:

```
ipsyscl = afeedback(ipsys, ipsysc);
```

The system now has eight states (four from the compensator, four from the original plant), three individual inputs (*u* and *y*, where *y* comprises ϕ and *x*) and three outputs (again, *u* and *y*). Here you have not added additional inputs to the system for process and measurement noise. Now you can

simulate the system's response to a slow sine input, starting with the cart at rest and the pendulum initially held in the upright ($\phi = 0$) position to obtain Figure 6-7:

```
t = 0:0.01:15;

u = pdm([sin(t/2); zeros(t);zeros(t)],t);ycl =
ipsyscl*u;

[outNames] = names(ycl);
```

Set plot attributes for all three plots:

```
p1=plot ({xlab = "Time",ylab = "Amplitude",
    columns = 1,rows = 3, hold})

for i = 1:3
    p1=plot (ycl(i,1), {graph_number = i,
        legend = outNames(i)});

endfor

plot(p1,{!hold})
```

Note The different y-axis scaling for each subplot is shown in Figure 6-7. This figure shows ϕ and x as a function of time, starting from zero, as a result of a sinusoidal force applied to the system input.

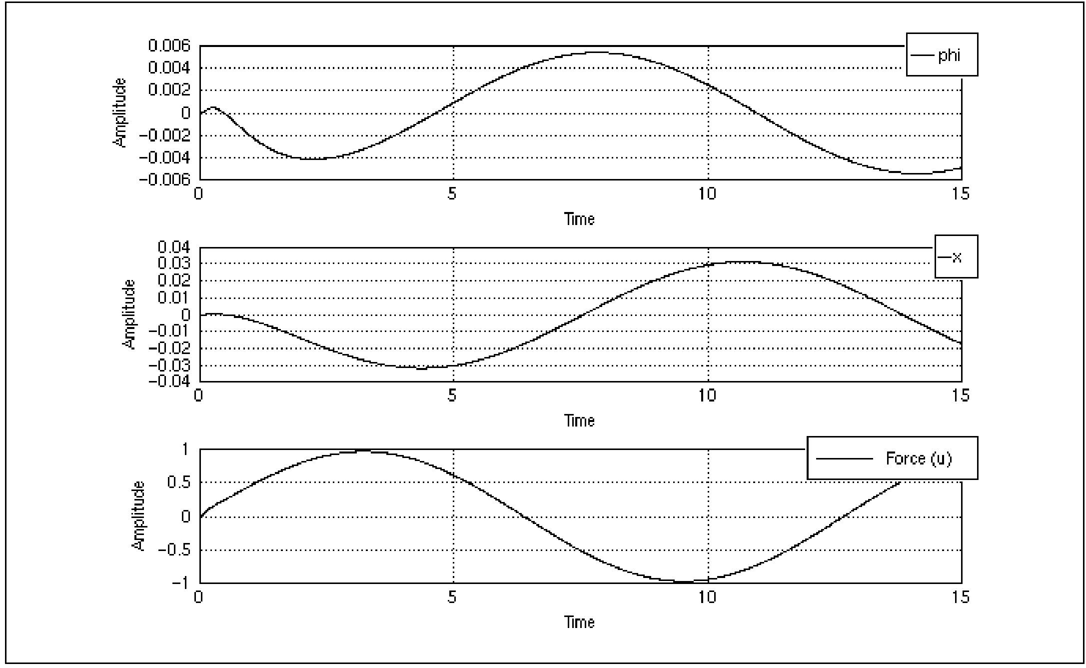


Figure 6-7. *f* and *x* as a Function of Time, Starting from Zero, as a Result of a Sinusoidal Force Applied to the System Input

Riccati Equation

Riccati equations, which take one of two distinct forms, arise in a number of linear systems and controls problems. The best-known use is in the solution of the optimal regulator and estimator problems, as described in the *Linear Quadratic Regulator* section and the *Linear Quadratic Estimator* section.

The continuous-time Riccati equation is given by:

$$A'P + PA - (PB + S) \times inv(R)(B'P + S') + Q = 0$$

The discrete-time Riccati equation is given by:

$$A'PA - P - (A'PB + S) \times inv(R + B'PB)(B'PA + S') + Q = 0$$

This function can be used to solve the optimal regulator problem, and by duality, the optimal estimator problem. An alternative form of the

continuous-time Riccati equation, which is used if *B* and *S* are not specified, is:

$$A'P + PA - PRP + Q = 0$$

Note The meaning of *R* is quite different in this case.

riccati( )

```
[P,resid, Kr, ev] = riccati(A,Q,R,{B,tol,S,d})
```

Here, `A` can be either a matrix or a system object.

If `A` is a matrix, `riccati( )` solves the continuous Riccati equation in the *Riccati Equation* section unless the B matrix is present; then it solves the discrete form (also shown in the *Riccati Equation* section).

If `A` is a system object, the solution method depends on whether the system is continuous or discrete. The B matrix is unnecessary to distinguish continuous from discrete.

The algorithms used are based on [Lau79] and [PLS80]. For the continuous case, an ordinary Schur solver is used. For the discrete-time case, the solution uses a generalized eigenvalue solver. For an example of a continuous Riccati equation, refer to Example 6-8. For an example of a discrete Riccati equation, refer to Example 6-9.

Example 6-8 Continuous Riccati Equation

$$A'P + PA - (PB + S)R^{-1}(B'P + S') + Q = 0$$

You can use `riccati( )` to find the Riccati solution and gain for the optimal regulator problem posed in Equation 6-9:

```
A = [0,1,0,0;21,0,0,0.8;0,0,0,1;0,0,0,-4];
B = [0,-2,0,1]';
Q = diagonal([1,0,0.1,0]);
R = B*((1e-5)\B');
Sys=system(A,B,rand(B'),[])

ys (a state space system) =

A
 0     1     0      0
21     0     0      0.8
 0     0     0      1
```

```
 0    0    0    -4
B
 0
-2
 0
 1
C
 0.211325    0.756044    0.000221135    0.330327
D
 0
X0
 0
 0
 0
 0
System is continuous
[P,resid] = riccati(A,Q,R);
norm(A'*P+P*A-P*R*P+Q,1)
ans (a scalar) = 2.53297e-13
R = 1e-5;
[P,resid]=riccati(Sys,Q,R);
norm(A'*P+P*A-P*B*inv(R)*B'*P +Q,1)
ans (a scalar) = 2.52492e-13
```

The small residue indicates that the problem was well posed and the solution is reliable.

Example 6-9 Discrete Riccati Equation

$$A'PA - P - (A'PB + S)(R + B'PB)^{-1}(B'PA + S') + Q = 0$$

```
A = [0,0,0;.3,-0.1,0;0,1,0];
Q = [2,0,0;0,0,0;0,0,10];
B = [1,0,0]';
RD = .25 ;
Sys=system(A,B,B',[],{dt=1})
Sys (a state space system) =
A
0       0      0
0.3    -0.1    0
```

```
0       1      0

B
1
0
0

C
1    0    0

D
0

X0
0
0
0

System is discrete, sampling at 1 seconds.

[P,resid]=riccati(Sys,Q,RD,B);

norm(A'*P*A-P-A'*P*B*inv(RD+B'*P*B)*B'*P*A+Q,1)

resid (a scalar) = 7.90593e-12

[P,resid]=riccati(Sys,Q,RD,B);
norm(A'*P*A-P-A'*P*B*inv(RD+B'*P*B )* B'*P*A+Q,1)

ans (a scalar) = 7.90593e-12
```

Steady-State System Response Using Lyapunov Equations

The Lyapunov family of matrix equations are used in a number of control design problems. The general continuous Lyapunov equation is

$$AX + XB = -C \tag{6-10}$$

The special form of the continuous Lyapunov equation replaces B with A':

$$AX + XA' = -C \tag{6-11}$$

These continuous Lyapunov equations have a unique solution X when $\lambda(A) + \lambda_j(B) \neq 0$ for any eigenvalues λ_i,λ_j, as proved in [Kai80]. This also means that for a stable continuous-time system, X will be unique because

all the eigenvalues of the system A matrix are negative. The discrete Lyapunov equation is:

$$AXA' + C = X \tag{6-12}$$

Analogously, the preceding equation has a unique solution X when $\lambda_i(A)\lambda_j(A) \neq 1$ for all i and j. Again, this means that a unique X exists for a stable discrete-time system matrix A, because all eigenvalues of A have absolute value less than 1 in this case.

You can use the Lyapunov equation to compute the state covariance matrix of a stable system with white noise input, as illustrated in [BH75]. For a continuous-time state-space system described by

$$\dot{x} = Ax + Bu \tag{6-13}$$

and supplied with zero-mean white noise $\omega(t)$ having covariance Q:

$$E[\omega(t)\omega'(\tau)] = Q\delta(t-\tau)$$

the state covariance $X = E[xx']$ is given by the differential Lyapunov equation:

$$\dot{X} = AX + XA' + BQB' \tag{6-14}$$

For the discrete-time system described by

$$x_{k+1} = Ax_k + Bu_k$$

the white noise input covariance is defined as in the continuous case, using a Kronecker rather than a Dirac delta.

For this case, the state covariance matrix X arises from the solution of the discrete Lyapunov equation:

$$X = AXA' + BQB' \tag{6-15}$$

After you have obtained the state covariance, you can obtain the output covariance Y easily. Whether you are using the following equation for the continuous case:

$$y = Cx + Du$$

or the following for the discrete case:

$$y_k = Cx_k + Du_k \qquad (6\text{-}16)$$

$$Y = CXC' + DQD'$$

These results derive from the Lyapunov method of stability analysis for linear systems. *Steady state* means that at some point the states no longer change. The derivative term $\dot{x}$ approaches zero in the large for continuous systems, and $x_{k+1} = x_k$ for discrete-time ones. The state value vector x for which this is true is defined as the equilibrium state. As stated in [Oga70], a unique equilibrium state exists for systems with a nonsingular A matrix, whereas infinitely many equilibrium states exist if A is singular. A system is described as asymptotically stable if the state values approach the equilibrium state over time, no matter what value of x one started with. Such systems will always satisfy the following: for any positive-definite matrix Q, a positive definite matrix X can be found satisfying in the X equation (Equation 6-15) for the continuous case and Y equation (Equation 6-16) for the discrete case.

Lyapunov equations also can be used to compute system controllability and observability grammians, which play an important role in internal balancing and model reduction. This application will be discussed further in the *Balancing a Linear System* section.

lyapunov( )

```
X = lyapunov(A,B,{C, discrete})
```

The `lyapunov( )` function provides a solution to both the discrete and continuous-time Lyapunov equations. When called with three inputs (A,B,C), it solves the general continuous Lyapunov equation (Equation 6-10); when called with two inputs (A,C), it solves the special Lyapunov equation (Equation 6-11). When called with two inputs (A,B) and the `{discrete}` keyword, it solves the discrete Lyapunov equation (refer to Equation 6-12). For examples of discrete, continuous, and special Lyapunov equation solutions, refer to Example 6-10.

Algorithm

The algorithm for `lyapunov( )` uses the Schur decomposition to convert A and B to upper triangular form, then finds the Lyapunov equation solution a column at a time by solving. `lyapunov( )` warns the user if the eigenvalues of (A + `eye`(A)) are close to –1, in which case singularity may occur and cause the function to terminate. Furthermore, if any combination

of the diagonal elements of the Schur-decomposed A and B matrices sum to zero, a warning is given that the continuous equation solution may not be unique. A similar warning appears for the discrete equation solution if the product of any of the eigenvalues is 1.

To solve the special Lyapunov equation, use the following syntax:

```
lyapunov(A,C)
```

Example 6-10 Lyapunov Equation Solutions

The following examples each give results close to zero.

Discrete Lyapunov Equation

$$A \times X \times A' + C = -X$$

```
A = [1, 2;-3,.4];

C = [-1,3;6,2];

X = lyapunov(A,C, {discrete})

X (a square matrix) =

-0.0829686  0.946549
 0.390993  -0.418771

norm(A*X*A'+C -X,1)

ans (a scalar) = 2.58127e-15
```

Continuous Lyapunov Equation

$$A \times X + X \times B = -C$$

```
A = [1,-3;2,5];
B = [-4,3;2,1];

C = [1,3;-6,2];
X = lyapunov(A,B,C)

X (a square matrix) =

 2.62963  -2.11111
-3.7037    2.22222

A*X + X*B + C;

norm(A*X + X*B + C,1)

ans (a scalar) = 0
```

Special Lyapunov Equation

$$A \times X + X \times A' = -C$$

```
A = [-4,10;2,7];
C = [.3,6;2,9];
X = lyapunov(A,C)

X (a square matrix) =

 1.1816   -0.209028
 1.12431  -0.773611

A*X + X*A' + C;
norm(A*X + X*A' + C,1)

ans (a scalar) = 5.4956e-15
```

rms( )

```
[Yrms,Ycov] = rms(Sys,Ucov)
```

The `rms( )` function computes the root-mean-square response (average power at the system output) and the output covariance of a dynamic system driven by zero-mean white noise input. You can specify the intensity of the noise with the optional input covariance parameter `Ucov`, which defaults to identity.

For a continuous system, the covariance of the states is given by X, where X is the differential Lyapunov solution (shown in Equation 6-14) with $\dot{X}$ equal to zero for steady-state. Thus, for a system with output Y defined by:

$$Y = Cx + Du$$

the output covariance matrix (Y_{cov}) is expressed as:

$$Y_{cov} = CXC' + DU_{cov}D'$$

The output covariance for a discrete system follows analogously, with X being the solution to Equation 6-12 in this case. Thus, a call to `lyapunov( )` forms the core of `rms( )`.

The diagonal terms of the covariance matrix correspond to the expected values of the squares of the power at each output. Taking the square root of these diagonal terms, you obtain the rms (root mean square) power at each output. For an example of `rms( )` responses, refer to Example 6-11.

Example 6-11 rms( ) Response

```
Sys = system([-2.3,0.01,5.1;0,-0.35,-2;
    0,2,-.35],[1,.25,.25]',[1.34,0,0],0);
w = logspace(0.01,1,50);
Uspec = pdm(ones(w),w);
[Ypsd,Yspec] = psd(Sys,Uspec);
```

Balancing a Linear System

Given a particular system model, the concept of model reduction centers on finding a lower-order model with similar input-output response characteristics. Typically this is assessed by comparing the impulse responses of the two systems [Moo81]. The goal in balancing a linear system is to find a state transformation that resolves the trade-off between controllability and observability, returning a transformed system whose states are equally controllable and observable. This raises the issue of quantifying a system's controllability or observability. You can do this by considering the system singular values associated with the mappings between the inputs and states, and those associated with the state-output mappings.

These singular values can be obtained from decompositions of two quantities referred to as the controllability and observability grammians. These quantities are represented by W_c and W_o respectively, and defined by the following equation for a system with an asymptotically stable A matrix.

$$W_c = \int_0^{\infty} e^{tA} BB' e^{tA'} dt$$

$$W_o = \int_0^{\infty} e^{tA'} C'C e^{tA} dt \qquad (6\text{-}17)$$

For continuous systems, the controllability and observability grammians satisfy the Lyapunov equations:

$$AW_c + W_cA' + BB' = 0$$

$$A'W_o + W_oA + C'C = 0 \qquad (6\text{-}18)$$

For discrete-time systems, the integrals in the W_c and W_o equations are replaced by summation signs and the grammians are obtained as the solutions of the discrete-time Lyapunov equations:

$$AW_cA' + BB' = W_c$$
$$A'W_oA + C'C = W_o \tag{6-19}$$

The controllability grammian must be full-rank for the system to be completely controllable; similarly, the observability grammian must be full-rank for the system to be completely observable (refer to [Kai80]). The condition number of W_c reflects how well conditioned the system model is with regard to pointwise state control.

The condition number of W_o reflects the condition of the model with regard to zero-input state-observation.

A linear transformation T of the system {A,B,C} also results in a linear transformation of the grammians. If the state vector is transformed as $x = T\hat{x}$, the system and grammian transformations in the following equations:

$$\hat{A} = T^{-1}AT \qquad \hat{W}_c = T^{-1}W_c(T')^{-1}$$
$$\hat{B} = T^{-1}B \qquad \hat{W}_o = T'W_oT$$
$$\hat{C} = CT$$
$$\hat{D} = D$$

Although the poles of the system (of the eigenvalues of A) do not change under the transformation, the singular values (eigenvalues of the grammians) do. However, the eigenvalues of the product of the grammians are invariant under transformation, and these are the singular values of the system input-to-state and state-to-output maps [LHPW87].

The system is defined as being internally balanced if for some transformation T, $\hat{W}_c^2 = \hat{W}_o^2 = \Sigma^2$

where

$$\Sigma^2 = diagonal([\sigma_1^2, \sigma_2^2, \ldots, \sigma_2^2])$$

and σ_1^2 through σ_n^2 are the singular values of the matrix H satisfying $\Sigma^2 = H'H$. They are termed the Hankel singular values. The σ_k^2 terms are ordered so that $\sigma_1^2 \geq \sigma_2^2 \geq \ldots \geq \sigma_n^2 \geq 0$.

The balanced system essentially gives the best compromise between how well conditioned the system is with regard to controllability and observability.

For model reduction problems, consider the balanced model partition as:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u$$

$$y = [C_1 C_2] \begin{bmatrix} x_1 \\ s_2 \end{bmatrix} + Du$$

$$\Sigma^2 = \begin{bmatrix} \Sigma_1^2 & 0 \\ 0 & \Sigma_2^2 \end{bmatrix}$$

with

$$\Sigma_1^2 = \mathrm{diag}onal(\sigma_1^2 .... \sigma_k^2)$$

$$\Sigma_2^2 = \mathrm{diag}onal(\sigma_{k+1}^2 .... \sigma_n^2)$$

The essence of a balanced model reduction is that if $\sigma^2_k >> \sigma^2_{k+1}$, the input/output behavior of the states in x_2 is much less important than that of the states in x_1. Eliminating the part of the model corresponding to x_2 will result in a reduced-order model which retains the most important input-output characteristics of the original system.

balance( )

```
[SysB,HSV,T] = balance(Sys)
```

The `balance( )` function performs input/output balancing on a linear system, returning the system transformed to a balanced form as `SysB`. `HSV` contains the second-order modes of the balanced system, or the singular values of `H`, where `H` is as defined previously.

T is the transformation relating the states of the original system to the states of the balanced system $x = T\hat{x}$. Transforming to balanced coordinates can be useful in model reduction because the relative importance of the state to the system's input/output performance is highlighted.

`balance( )` first finds the controllability and observability grammians using `lyapunov( )` for both the discrete and continuous cases. It then performs a singular-value decomposition of both grammians:

```
[Sc,Uc,Vc]=SVD(Wc); [So,Uo,Vo]=SVD(Wo)
```

and constructs the H matrix from the square roots of the singular values:

```
H=diagonal(sqrt(So))*Uo'*Uc*diagonal(sqrt(Sc))
```

The singular-value decomposition of H returns the Hankel singular values `HSV`. The transformation matrix T is obtained by backsolving and retransforming. The algorithm is given in [Moo81]. When the transformation has been found, the balanced system matrices then can be obtained from the original system through Equation 6-18. For an example of how to balance a system, refer to Example 6-12.

Example 6-12 Balancing a System

Taking a hypothetical system:

```
A=-[1,2,3,4;0,5,6,7;0,0,8,9;0,0,0,10];
B=[0;0;0;1];
C=[1,0,0,0];
D=0;
Sys = system(A,B,C,D);
```

Computing the controllability and observability grammians and noting their rather high condition numbers:

```
Wc=lyapunov(A,B*B');
Wo=lyapunov(A',C'*C);
condition(Wc)

ans (a scalar) = 283.029

condition(Wo)

ans (a scalar) = 1112.66
```

You then balance the system,

```
[SysB,HSV,T]=balance(Sys);
[Ab,Bb,Cb]=ABCD(SysB);
```

and compare the condition numbers of the balanced system's grammians:

```
WcB=lyapunov(Ab,Bb*Bb');
WoB=lyapunov(Ab',Cb'*Cb);
condition(WcB)

ans (a scalar) = 12.7394

condition(WoB)

ans (a scalar) = 12.7394
```

The condition numbers are now much smaller, and they are equal, indicating that the system is now equally well conditioned in terms of its controllability and observability.

Modal Form of a System

The modes of a state-space system are defined as corresponding to the eigenvalues of the system's A matrix. The modes of a system are distinct from the states of a system; because a given system can be arbitrarily transformed, the states can be arbitrarily assigned. The modes, on the other hand, do not change from realization to realization of a given system.

The modal decomposition of a system can be obtained mathematically through a Laplace transform, partial fraction decomposition, and eigen decomposition as shown in [Kai80]. The key advantage of a modal decomposition is that it provides a means by which large systems can be represented as a parallel combination of first-order systems. In addition, the modal decomposition of a given system representation is often better conditioned numerically.

The modal form is particularly useful with structured dynamic systems whose poles primarily occur as complex pairs. When a system model has been converted to modal form, it can be reduced to focus attention on the particular modes whose dynamics are of interest.

modal( )

```
[SysMod, T] = modal(Sys)
```

The `modal( )` function uses eigenvalue decomposition to find the Jordan form of the system matrix A (all eigenvalues on the diagonal). This approach is appropriate for models without repeated eigenvalues; modal decomposition of a system with repeated eigenvalues is numerically unreliable. If a system with repeated or very closely spaced eigenvalues is passed to `modal( )`, a warning appears noting that the results may not be

accurate. Given a variable `Sys` built from the matrices {A,B,C,D}, the modal decomposition `SysMod` is built from $T^{-1}AT$, $T^{-1}B$, CT, and D, where T is the transformation matrix to modal form. If you have complex poles, then $T^{-1}AT$ is in block diagonal form. Initial conditions X_0 also are transformed to $T^{-1}X_0$.

`modal( )` does not accept input systems in transfer-function form, as the concept of modes applies only to a state-variable system representation and modes and poles are not interchangeable terms. The poles of a transfer function always correspond to the system modes (eigenvalues of the system A matrix).

mreduce( )

```
SysRed = mreduce(Sys, keep)
```

The `mreduce( )` function computes a reduced-order form of a given system by retaining the states indicated within the vector `keep`. States not specified within this vector are eliminated to obtain a lower-order model `SysRed`.

`mreduce( )` is implemented by partitioning the state vector x into two subvectors, x_1 (states to be retained in the reduction) and x_2 (states to be eliminated in the reduction), so that:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Similarly, the A, B, and C matrices are partitioned according to this state partition:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \qquad C = \begin{bmatrix} C_1 & C_2 \end{bmatrix}$$

The model reductions differ for the continuous and discrete-time cases because the updates for the states being eliminated are handled differently in the respective differential and difference equations. In both cases, the eliminated states are taken to be constant over time. In the continuous case,

the derivative of x_2 is set to zero, resulting in reduced-order state equations of the form:

$$\dot{x}_1 = (A_{11} - A_{12}A_{22}^{-1}A_{21})x_1 + (B_1 - A_{12}A_{22}^{-1}B_2)u$$
$$y = (C_1 - C_2A_{22}^{-1}A_{21})x_1 + (D - C_2A_{22}^{-1}B_2)u$$

In the discrete case, x_{2k+1} is taken to be equal to x_{2k} so that the state equations become:

$$x_{1k+1} = [A_{11} - A_{12}(A_{22} - I)^{-1}A_{21}]x_{1k} + [B_1 - A_{12}(A_{22} - I)^{-1}B_2]u_k$$

$$y = [C_1 - C_2(A_{22} - I)^{-1}A_{21}]x_{1k} + [D - C_2(A_{22} - I)^{-1}B_2]u_k$$

When using `mreduce( )`, remember to remove states corresponding to complex conjugate poles. Not doing so—that is, eliminating only one pole in a pair—will produce a meaningless system.

More complex model reduction algorithms, which are intended to model complete system dynamics in the absence of one of more states, are available with the Xmath Model Reduction Module, as shown in Figure 6-8 and in Example 6-13.

Example 6-13 Model Reduction Module

```
A= [0.37,0.26,0.22,0.67;
    0,0.52,0.63,0.20;
    0,0,0.76,0.39;
    0,0,0.04,0.83]
B = [0,1.7e-5,0,0.0004]'
C = [1,0,1,0]
D = 0

Sys = system(A,B,C,D,{dt = 0.2});
[SysM, T] = modal(Sys)

SysM (a state space system) =

A
0.37 0 0 0
0 0.52 0 0
0 0 0.665289 0
```

```
0 0 0 0.924711

B
-0.00116788
 0.00272531
 0.00334243
-0.00162497

C
1 0.866186 -0.848754 -1.0118

D
0

X0
0
0
0
0

State Names
-----------
State 1 State 2 State 3 State 4

Input Names
-----------
Input 1

Output Names
------------
Output 1

System is discrete, sampling at 0.2 seconds.

T (a square matrix) =

1 0.866186 -0.668844 -0.641745
0 0.499722 -0.71998 -0.653294
0 0 -0.17991 -0.370059
0 0 0.043691 -0.15629

eig(A)

ans (a column vector) =

0.37
0.52
0.665289
0.924711

T\A*T

ans (a square matrix) =

0.37 5.55112e-17 -6.245e-17 -2.77556e-16
```

```
0 0.52 1.94289e-16 6.66134e-16
0 0 0.665289 4.44089e-16
0 0 0 0.924711

SysMR = mreduce(SysM, [1,2,4])

SysMR (a state space system) =

A
0.37 0 0
0 0.52 0
0 0 0.924711

B
-0.00116788
 0.00272531
-0.00162497

C
1 0.866186 -1.0118

D
-0.00847566

X0
0
0
0
0

State Names
-----------
State 1 State 2 State 4

Input Names
-----------
Input 1

Output Names
------------
Output 1

plot(step(SysM, 0:.2:10))
plot(step(SysMR, 0:.2:10),{keep,
   legend=["Original System";"Reduced System"]})
```

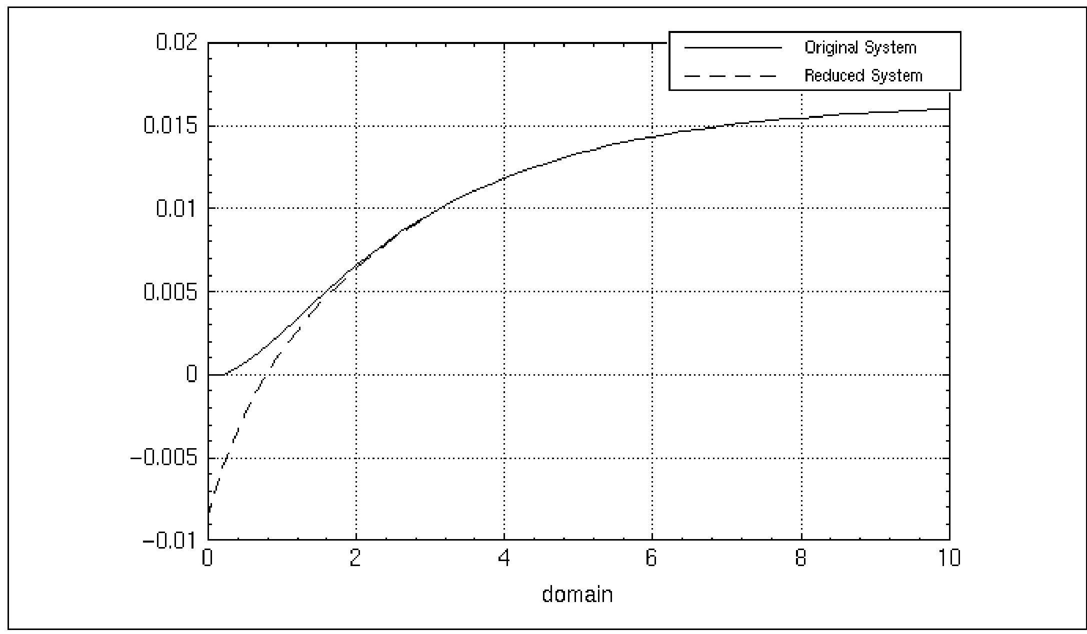
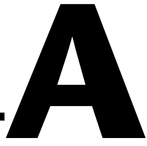


Figure 6-8. Modal System and Reduced Modal System



Technical References

[BeV88] T. Beelen, P. Van Dooren, "An improved algorithm for the computation of Kronecker's canonical form of a singular pencil," *Linear Algebra and Applications*, 105, pages 9–65, 1988.

[BH75] A.E. Bryson and Y.C. Ho, *Applied Optimal Control*, Hemisphere Publishing Corporation, Washington, D.C., 1975.

[ChB84] R.V. Churchill and J.W. Brown, *Complex Variables and Applications*, McGraw-Hill Book Company, New York, 1984.

[DeS74] C.A. Desoer and J.D. Schulman, "Zeros and Poles of Matrix Transfer Functions and Their Dynamical Interpretation," *IEEE Transactions on Circuits and Systems*, CAS-21, pages 3–8, 1974.

[EmV82] A. Emami-Naeini and P. Van Dooren, "Computation of Zeros of Linear Multivariable Systems," 18, 4, pages 415–430, 1982.

[FPE87] G.F. Franklin, J.D. Powell, A. Emami-Naeini, *Feedback Control of Dynamic Systems*, Addison-Wesley Publishing Company, New York, 1987.

[FPW90] G.F. Franklin, J.D. Powell, M.L. Workman, *Digital Control of Dynamic Systems*, Addison-Wesley Publishing Company, New York, 1990.

[GrD86] R.M. Gray and L.D. Davisson, *Random Processes, A Mathematical Approach for Engineers*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[HW91] P. Hsu and J. Wendlandt, "The Wedge-A Controller Design Experiment," Preprints, IFAC Advances in Control Education, pages 169–174, 1991.

[KaB61] R.E. Kalman and R. Bucy, "New Results in Linear Filtering and Prediction," *Transactions of the American Society of Mechanical Engineers*, 83D, page 95, 1961.

[Kal60] R.E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the American Society of Engineers*, 82D, page 35, 1960.

[Kai80] T. Kailath, *Linear Systems*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.

[Kai81] T. Kailath, *Lectures on Wiener and Kalman Filtering*, Springer-Verlag, New York, 1981.

[Lau79] A.J. Laub, “A Schur method for solving algebraic Riccati equations,” *IEEE Transactions on Automatic Control*, AC-24, pages 913–621, 1979.

[Lau86] A.J. Laub, “Efficient Calculation of Frequency Response Matrices from State Space Models,” *ACM Transactions on Mathematical Software*, 12, 1, pages 26–33, 1986.

[Leo89] A. Leon-Garcia, *Probability and Random Processes for Electrical Engineering*, Addison-Wesley Publishing Company, New York, 1989.

[LHPW87] A.J. Laub, M.T. Heath, C.C. Paige, and R.C. Ward, “Computation of System Balancing Transformations and Other Applications of Simultaneous Diagonalization Algorithms,” *IEEE Transactions on Automatic Control*, AC-32, 2, pages 115–121, 1987.

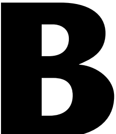
[Oga70] K. Ogata, *Modern Control Engineering*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1970.

[PLS80] T. Pappas, A.J. Laub, and N.R. Sandell, Jr., “On the Numerical Solution of the Discrete-Time Algebraic Riccati Equation,” *IEEE Transactions on Automatic Control*, AC-25, 4, pages 631–641, 1980.

[ShH92] B. Shahian and M. Hassul, *Control System Design Using MATRIXx*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[Van79] P. Van Dooren, “The computation of Kronecker’s canonical form of a singular pencil,” *Linear Algebra and Applications*, 27, pages 103–141, 1979.

[Van81] P. Van Dooren, “The generalized eigenstructure problem in linear system theory,” *IEEE Transactions on Automatic Control*, AC-26, 1, pages 111–129, 1981.



Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at `ni.com` for technical support and professional services:

- **Support**—Online technical support resources at `ni.com/support` include the following:
  - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Discussion Forums at `ni.com/forums`. National Instruments Application Engineers make sure every question receives an answer.

    For information about other technical support options in your area, visit `ni.com/services` or contact your local office at `ni.com/contact`.
- **Training and Certification**—Visit `ni.com/training` for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.

System Integration—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit `ni.com/alliance`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch

office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Index

A

abcd, 1-16, 2-8
adjoint system, 3-3
afeedback, 3-4
append, 3-6
appending dynamic systems, 3-6
autocorrelation function, 5-20

B

balance, 6-35
bilinear transform, 2-14
Bode
  default frequency range (deffreqrange), 5-10
  format, 5-8
  frequency analysis, 5-7
  plots, 5-7
bode, 5-10

C

cancel, 2-2
cascaded systems, 5-21
Cauchy's principle, 5-15
caution, A-1
check, 1-5, 4-6
  convert keyword, 2-12
  with system objects, 2-12
choosing a sample rate, 1-18
closed-loop system
  eigenvalues, 6-14
  simulate performance, 1-23
combinepf, 4-9
compensator
  direct, in predictor form, 6-22
  LQG, 6-21
connect, 3-8
connection, 3-1
  parallel, 3-1, 3-2
  series, 3-2
  using * operator, 1-16, 3-2
constant gain feedback, 3-8
constant magnitude and phase loci, 5-14
continuous equivalent to a discrete system, 2-18
continuous system, 4-13
  analysis, 2-7
  checking for, 2-12
continuous time Riccati equation, 6-13, 6-25
controllability, 6-4, 6-35
  grammians, 6-30
  matrix, 6-2
controllable, 6-3
controllable partition of a state-space system, 6-3
conventions used in the manual, *iv*
coordinate transformation, 6-7
corner frequency, 5-12
cross-spectral density, 5-20

D

decibel gain, 5-8, 5-16
deffreqrange, 5-10
deftimerange, 4-15
delay time, td, 4-18
delta
  function, 6-17
  impulse, 5-20
diagnostic tools (NI resources), B-1
discrete
  Riccati equation, 6-13, 6-20
  system, 2-4, 4-13
    analysis, 2-7
    checking for, 2-12

converting to continuous equivalent, 2-19
from a continuous system (example), 1-18
discrete-time Riccati equation, 6-25
discretize, 2-13
discretizing a system
backward rectangular method, 2-14
forward rectangular method, 2-14
hold equivalence methods, 2-15
pole-zero matching, 2-15
trapezoid method, 2-14
Tustin's method, 2-14
using ztransform for zero-order hold, 2-15
with exponential keyword, 2-16
documentation
conventions used in the manual, *iv*
NI resources, B-1
domain, regular, 4-11
drivers (NI resources), B-1
duality, 6-6, 6-9, 6-10
principle, 6-14
dynamic system, 6-32
appending systems, 3-6
frequency response, 5-5
improper, 3-4
impulse response, 4-13
indexing, 2-12
size, 2-12

E

eigenvalues, 4-2, 6-6
generalized solver, 6-26
encirclements, 5-15
equilibrium state, 6-30
error covariance, 6-19
estimator, 6-5, 6-16, 6-20
optimal, 6-18
estimator system, 1-17
examples (NI resources), B-1
expectation operator, 6-17
exponential discretization method, 1-19, 2-16

F

feedback, 3-11
constant gain, 3-8
full-state, 1-13, 1-22
loop, 3-11
single input syntax, 1-9
using second input to create negative feedback loop, 1-10
filters, 5-20
Fourier transform, 5-20
freq, to find values of a transfer function at one frequency, 1-10
frequency response, 1-19, 5-5
calculating, 5-5
open-loop, 5-14
full-state feedback, 1-13, 1-22

G

gain margin, 5-8, 5-9, 5-12
general interconnection around a system, 3-8
generalized eigenvalue solver, 6-26
grammians, 6-30, 6-33
graphics window, 5-3

H

helicopter hover problem, 1-4
ad hoc approach, 1-4
discrete formulation, 1-4, 1-18
state feedback and observer design, 1-4, 1-13
help, technical support, B-1

I

impulse, 4-13
- input, 4-13
  - continuous time, 4-13
  - discrete time, 4-13
- response, 4-13, 6-33

initial, 4-16
initial conditions, 4-14, 4-16
input
- disturbance matrix, 6-17
- names, extracting, 2-11

instability, 5-9
instrument drivers (NI resources), B-1
internally balanced system (definition), 6-34
inverse, 3-3
inverted pendulum problem, 6-21

K

KnowledgeBase, B-1
Kronecker delta function, 6-29

L

linear
- quadratic Gaussian compensation, 6-21
- quadratic regulator, 6-12
- systems, defining, 2-1
- transformation, 6-34

logarithmic plots, 5-8
LQG
- compensator, 6-21
- estimator, weighting matrix, 1-21
- regulator, weighting matrix, 1-21

lqgcomp, 6-23
Lyapunov
- continuous equation, 6-28
- discrete equation, 6-29

lyapunov, 6-30

M

magnitude, 5-5, 5-8
makecontinuous, 2-17
- verifying discretization with, 2-18

makepoly, 2-3
margin, 5-12
Markov parameters, 4-13
matrix
- controllability, 6-2
- inputs disturbance, 6-17
- observability, 6-5
- Riccati equation, 6-18
- transformation, 6-9

MATRIXx help, 1-3
maximum overshoot, Mp, 4-18
measurement
- noise, 6-16, 6-17
- update, 6-19

MIMO systems, 2-5
minimal, 1-11, 6-8
minimal realization of a system, 6-7
modal, 6-37
modern control, 6-1
modes of a system, 6-37
mreduce, 6-38

N

names, 2-11
- default for systems, extracting, 2-11
- modifying state estimate, 1-15

National Instruments support and services, B-1
neutral stability, 5-9
nichols, 5-14
noise
- intensity matrices, 6-17
- measurement, 6-16, 6-17, 6-23
- process, 6-16

steady-state response to white noise, 6-29
white process, 6-23
nomenclature, 1-2
numden, 2-10
numerical integration methods, 2-14
Nyquist
contour, 5-19
plot, 5-17
stability criterion, 5-15
nyquist, 5-16

O

observability, 6-4, 6-35
grammians, 6-30
matrix, 6-5
observable, 6-6
observable partition of a state-space system, 6-6
observer, 6-5, 6-16
-based controller, 1-14
gains, finding with poleplace, 1-15
open-loop
frequency response, 5-14
poles, 5-15
operators
linear system interconnection, 3-1
overloaded, 3-1
optimal
estimator, 6-18
regulator, 6-12
output
covariance, 6-32
names, extracting, 2-11

P

parallel connection, 3-1, 3-2
partial fraction expansion, 4-9
PDM, 4-10, 5-6
peak time, tp, 4-18
period, 2-10
phase, 5-5, 5-11
f, 5-8
margin, 5-8, 5-9, 5-12
rate of change, 5-12
tracking, 5-6
poleplace, 6-10
finding feedback gains, 1-14
finding observer gains, 1-15
poles, 1-1, 2-2, 4-3
open-loop, 5-15
placement, 6-1
problem, 6-10
unstable open-loop, 5-15
pole-zero
cancellation, 1-11
matching, 2-15
power spectral density, 5-20, 5-21
process noise, 6-16
programming examples (NI resources), B-1
psd, 5-20

Q

quadratic performance index, 6-12, 6-13

R

regular domain, 4-11
regulator
linear quadratic, 6-12
optimal, 6-12
residue, 1-2, 4-8
residues, definition, 4-5
Riccati
continuous-time equation, 6-25, 6-26
discrete-time equation, 6-25, 6-27
riccati, 6-26
Riccati discrete-time equation, 6-27
rise time, tr, 4-18
rlocus, 1-7, 5-3

rms, 6-32
root locus, plotting, 5-3
root-mean-square response, 6-32

S

sample period, extract with period, 2-10
sample rate, 2-4
  choosing, 1-18
scale system output, 1-16
Schur
  decomposition, 6-30
  solver, 6-26
selection matrix for adding or removing inputs, 3-9
sensitivity of states, determining, 4-17
serial, 3-1
series connection, 3-2
  * operator, 1-16, 3-2
settling time, ts, 4-18
simulate performance of a closed-loop system, 1-23
singular values, 6-35
singular-value decomposition, 6-36
SISO systems, 2-3
software (NI resources), B-1
square system, 3-5
stability
  checking for, 2-12
  criteria, 5-8
  criterion for Nyquist, 5-15
  neutral, 5-9
stair, 6-9
staircase
  algorithm, 6-3, 6-6
  form, 6-9
state
  covariance, 6-29
  names, extracting, 2-11
  sensitivity, determining, 4-17
  transformation, 6-33
state-space system, 2-5
  checking for, 2-12
  controllable partition of, 6-3
  convert to transfer function form, 1-5
  decompose with abcd, 2-8
  discrete, creating, 2-5
  extracting transfer function polynomials with numden, 2-10
  modes of, 6-37
  observable partition of, 6-6
steady state, 6-30
  system response, 6-28
step, 1-9, 4-18
step response, 4-18
support, technical, B-1
system, 2-6
  analysis, 2-7
  cascaded, 5-21
  connections, 3-1
  continuous, 4-13
  controllability, 6-4
  controllable, 6-1
  discrete, 4-13
  general interconnection, 3-8
  impulse input to, 4-13
  initial values for states, 2-7
  input names, 2-7
  inverse, 3-3
  keywords, 2-7
  minimal, 6-1
  objects, using check with, 2-12
  observability, 6-4
  observable, 6-1
  output names, 2-7
  reformat an existing system, 2-8
  square, 3-5
  states names, 2-7

T

technical support, B-1
time
  domain simulation, general, 4-10
  update, 6-19
training and certification (NI resources), B-1
transfer function
  checking for, 2-12
  coefficients form, 2-3
  converted to state space before decomposition, 2-9
  creating, 2-3
  form, definition, 4-2
  formed from partial fractions, 4-9
  pole-zero-gain form, 2-4
  polynomials, extracting from state-space system, 2-10
  system models, 2-2
  variable, 2-3
transform
  bilinear, 2-14
  Fourier, 5-20
  trapezoidal, 2-14
transformation, 6-9
  coordinate, 6-7
  linear, 6-34
  matrix, 6-9
  state, 6-33
transmission zeros, 4-4
trapezoid method for discretization, 2-14
troubleshooting (NI resources), B-1
Tustin's discretization method, 2-14

U

uncontrollable modes, 6-7
  checking for, 1-21
unity-gain negative feedback, 1-9
unobservable modes
  checking for, 1-21
unobservable uncontrollable modes, 6-7
unstable open-loop poles, 5-15

W

Web resources, B-1
wedge problem, 1-20
weighting, 6-13
  matrices, 6-13, 6-20
    for LQG regulator/estimator, 1-21
white noise, 6-23

Z

zeros, 1-2, 4-3
  transmission, 4-4
  zeros of the transfer function, 2-2